



普通高等教育“十一五”国家级规划教材

高等院校信息技术规划教材

微型计算机原理 与接口技术

孙力娟 李爱群 仇玉章 陈燕俐 周宁宁 编著

INFORMATION TECHNOLOGY
INFORMATION TECHNOLOGY
INFORMATION TECHNOLOGY



清华大学出版社



普通高等教育“十一五”国家级规划教材

高等院校信息技术规划教材

微型计算机原理 与接口技术

孙力娟 李爱群 仇玉章 陈燕俐 周宁宁 编著

清华大学出版社
北京

内 容 简 介

本书以 32 位微处理器为背景,讲述微型计算机原理、汇编语言程序设计和接口技术。内容主要包括: Pentium 微处理器内部结构、x86 基本指令和多媒体指令、汇编语言程序设计、总线概念及微型计算机系统典型总线、存储系统、输入/输出系统、中断系统、串行通信和并行接口、DMA 传送、数模和模数转换、保护模式下的程序设计和 Win32 汇编语言程序设计等。

本书可作为高等院校计算机专业及电类相关专业本科生微型计算机原理及应用、汇编语言程序设计、微型计算机接口技术及微型计算机原理与接口技术等课程的教材和参考书。通过删减适当章节,也适合非电类专业微型计算机原理及应用和微型计算机原理与接口技术等课程的教学,同时也可供自学者及从事计算机应用的工程技术人员参考。

本书封面贴有清华大学出版社防伪标签,无标签者不得销售。

版权所有,侵权必究。侵权举报电话: 010-62782989 13501256678 13801310933

图书在版编目(CIP)数据

微型计算机原理与接口技术 / 孙力娟等编著. —北京: 清华大学出版社, 2007. 2
(高等院校信息技术规划教材)

ISBN 978-7-302-14195-2

I. 微… II. 孙… III. ①微型计算机—理论 ②微型计算机—接口 IV. TP36

中国版本图书馆 CIP 数据核字(2006)第 142411 号

责任编辑: 马瑛珺 刘 霞

责任校对: 梁 毅

责任印制: 孟凡玉

出版发行: 清华大学出版社

<http://www.tup.com.cn>

c-service@tup.tsinghua.edu.cn

社 总 机: 010-62770175

投稿咨询: 010-62772015

地 址: 北京清华大学学研大厦 A 座

邮 编: 100084

邮购热线: 010-62786544

客户服务: 010-62776969

印 刷 者: 北京密云胶印厂

装 订 者: 北京市密云县京文制本装订厂

经 销: 全国新华书店

开 本: 185×260 印 张: 29.75 字 数: 706 千字

版 次: 2007 年 2 月第 1 版 印 次: 2007 年 2 月第 1 次印刷

印 数: 1~3000

定 价: 38.00 元

本书如存在文字不清、漏印、缺页、倒页、脱页等印装质量问题,请与清华大学出版社出版部联系调换。联系电话: 010-62770177 转 3103 产品编号: 021201-01

前言

Foreword

微型计算机原理与接口技术是理工类学生学习和掌握微型计算机基本组成、工作原理、接口技术以及汇编语言程序设计的重要课程。通过本课程的学习,能够使学生具有微型计算机系统软硬件开发和应用的基本能力。

为了适应微型计算机发展水平,本书以 32 位微处理器 Pentium 作为背景,讲述微型计算机原理、汇编语言程序设计和接口技术。随着 Windows、Linux 等多任务操作系统逐渐成为当前主流操作系统,本书增加了保护模式及 Win32 汇编语言程序设计等方面内容,有一定深度,并具有较强的实用性。

全书共有 15 章。

第 1 章为基础理论部分。包括数制、码制等基础知识,计算机组成以及微型计算机的发展历史。

第 2 章以 Pentium 为代表,介绍 32 位微处理器的结构、微处理器各部件的功能、引脚信号定义、工作模式,并对典型总线操作时序进行分析。

第 3 章介绍 80x86 的寻址方式和指令系统。

第 4 章介绍汇编程序的开发过程和伪指令。

第 5 章介绍实模式下汇编语言编程格式,并通过程序实例说明汇编语言程序设计的基本方法,包括分支、循环、子程序调用、代码转换、宏指令以及模块化程序设计等。

第 6 章介绍总线的概念、微型计算机系统中常用的总线标准和 32 位微型计算机总线结构。

第 7 章介绍微型计算机的存储器系统,包括存储器的概念、分类、存储器件以及和微处理器的连接等。

第 8 章介绍微型计算机系统中的输入/输出及 8254 定时/计数器及其应用。

第 9 章介绍微型计算机中断系统、中断控制器 8259A 和中断程序设计。

第 10 章介绍微型计算机系统串行通信及其应用。

第 11 章介绍并行接口芯片 8255A 及打印机接口。

第 12 章介绍 DMA 传送及 8237A DMA 控制器。

第 13 章介绍数模和模数转换原理以及接口芯片的功能。

第 14 章介绍 32 位微处理器保护模式下的运行机制及其编程方法。

第 15 章介绍 Win32 汇编语言的基础知识、Win32 汇编源程序的格式以及用 Win32 汇编语言编写 Windows 窗口程序的方法。

书中含有大量的程序实例,所有实例都经过上机验证,每章后均有习题。

本书第 1 章、第 2 章和第 6 章由李爱群老师编写;第 3 章、第 4 章、第 5 章、第 9 章和第 13 章、第 8.4 节、10.3 节、10.4 节和 11.2 节由仇玉章老师编写;第 7 章由周宁宁老师编写;第 8 章、第 10 章、第 11 章和第 12 章由孙力娟老师编写;第 14 章、第 15 章和 3.5 节由陈燕俐老师编写,并完成全书的整理。

在本书的编写过程中,洪龙老师、邓玉龙老师和薛明老师提出了许多宝贵建议,在此向他们表示衷心感谢!

由于编者水平有限,书中难免有错漏之处,恳请读者和同行批评指正。

编 者

2005 年 12 月

目录

contents

第 1 章 计算机基础	1
1.1 计算机中的数制	1
1.1.1 常用计数制	1
1.1.2 数制转换	2
1.2 计算机中数据的编码	3
1.2.1 数值数据的编码与运算	3
1.2.2 字符的编码	6
1.3 浮点数基本概念	7
1.3.1 浮点数	8
1.3.2 浮点机器数	8
1.3.3 浮点数的数值范围	9
1.4 计算机系统的基本组成	10
1.4.1 计算机系统的硬件组成	10
1.4.2 计算机系统的软件组成	11
1.4.3 微型计算机的硬件结构	12
1.4.4 微型计算机的分类和发展	13
习题	15
第 2 章 80x86 微处理器	16
2.1 Intel 微处理器发展简况	16
2.2 32 位微处理器内部结构	17
2.2.1 Pentium 内部结构	18
2.2.2 Pentium 微处理器结构特点	19
2.2.3 32 位微处理器的编程结构	20
2.3 32 位微处理器的外部引脚	27
2.4 32 位微处理器的工作模式	31
2.4.1 80x86 的地址空间	32

2.4.2	实地址模式	32
2.4.3	保护虚拟地址模式介绍	34
2.4.4	虚拟 8086 模式介绍	35
2.5	32 位微处理器的典型时序	36
2.5.1	时钟周期、总线周期和指令周期	36
2.5.2	Pentium 总线周期的时序分析	36
习题	38
第 3 章	指令系统	39
3.1	概述	39
3.1.1	指令的书写格式	39
3.1.2	符号指令的书写格式	39
3.2	80486 寻址方式	40
3.2.1	立即寻址	40
3.2.2	寄存器寻址	41
3.2.3	存储器操作数的寻址方式	41
3.2.4	80486 寻址方式的段约定和段超越	45
3.3	80486 标志寄存器	46
3.4	80486 基本集指令	49
3.4.1	传送类指令	49
3.4.2	算术运算指令	53
3.4.3	转移和调用指令	62
3.4.4	逻辑运算和移位指令	68
3.4.5	串操作指令	71
3.4.6	处理机控制指令	78
3.5	80x86 多媒体指令	79
3.5.1	MMX 指令	79
3.5.2	SSE 指令	84
习题	85
第 4 章	宏汇编语言	87
4.1	汇编语言程序的开发过程	87
4.2	汇编源程序的语句类型	88
4.3	宏汇编基本语法	89
4.3.1	标号、变量和常量	89
4.3.2	运算符	90
4.4	数据定义伪指令	92

4.5 宏汇编语言基本语句	95
习题	101
第5章 汇编语言程序设计	102
5.1 汇编源程序的编程格式	102
5.1.1 EXE 文件的编程格式	102
5.1.2 COM 文件的编程格式	103
5.1.3 EXE 文件和 COM 文件的内存映像	104
5.1.4 程序段前缀	105
5.1.5 返回 DOS 的其他方法	106
5.1.6 源程序堆栈段的设置	108
5.2 DOS 系统 I/O 功能调用	108
5.3 BIOS 键盘输入功能调用	113
5.4 文本方式 BIOS 屏幕功能调用	114
5.4.1 显示器	114
5.4.2 文本方式 BIOS 屏显功能调用	116
5.5 分支程序	118
5.6 循环程序	121
5.7 子程序及其调用	123
5.8 宏指令与条件汇编	128
5.8.1 宏指令与宏调用	128
5.8.2 条件汇编	131
5.9 代码转换	132
5.10 数值计算和数据处理	140
5.11 字符串的动态显示技术	154
5.12 模块化程序设计	157
5.12.1 支持模块化程序的伪指令	158
5.12.2 模块化程序的设计考虑	158
5.12.3 模块化程序设计举例	159
5.12.4 宏指令共享	164
习题	167
第6章 总线	168
6.1 总线基本概念	168
6.1.1 总线的类型与总线结构	168
6.1.2 总线的性能	169
6.1.3 总线信息的传送方式	170



6.2	典型总线标准	171
6.2.1	AT 总线	171
6.2.2	PCI 总线	174
6.3	通用外部总线标准	179
6.3.1	并行 I/O 标准接口 IDE 和 EIDE	180
6.3.2	并行 I/O 标准接口 SCSI	180
6.3.3	通用串行总线 USB	181
6.3.4	视频接口 AGP	187
6.4	32 位微型计算机总线结构	188
习题	190
第 7 章	存储器系统	191
7.1	概述	191
7.1.1	存储系统概念	191
7.1.2	存储器的体系结构	192
7.1.3	存储器的分类	194
7.1.4	存储器的主要性能指标	195
7.2	随机存储器与只读存储器	197
7.2.1	RAM 的分类与常用 RAM 芯片的工作原理	197
7.2.2	ROM 的分类与常用 ROM 芯片的工作原理	203
7.3	微型计算机系统中的存储器组织	206
7.3.1	存储器的扩展技术	206
7.3.2	CPU 与主存储器的连接	210
7.3.3	PC 机的存储器组织	213
习题	217
第 8 章	输入/输出系统	219
8.1	概述	219
8.1.1	接口电路	219
8.1.2	输入/输出端口	220
8.1.3	输入/输出指令	221
8.2	微型计算机系统与输入/输出设备的信息交换	222
8.2.1	无条件传送方式	222
8.2.2	查询方式	223
8.2.3	中断控制方式	225
8.2.4	直接存储器存取方式	225
8.3	可编程定时器/计数器 8254	226

8.3.1	8254 的内部结构	226
8.3.2	8254 引脚功能	228
8.3.3	8254 的工作方式	229
8.3.4	8254 的控制字与编程方法	234
8.3.5	8254 在微型计算机系统中的应用	237
8.4	发声系统与音乐程序设计	239
8.4.1	PC 系列机发声系统	239
8.4.2	音乐程序设计举例	241
习题	244
第 9 章	中断系统	245
9.1	中断的基本概念	245
9.2	80x86 中断指令	246
9.3	中断向量	247
9.4	微型计算机系统的中断分类	250
9.4.1	CPU 中断	250
9.4.2	软件中断	251
9.5	8259A 中断控制器	253
9.5.1	8259A 内部结构	253
9.5.2	8259A 中断管理方式	255
9.5.3	8259A 初始化	258
9.6	微型计算机系统可屏蔽中断	263
9.6.1	可屏蔽中断与非屏蔽中断	263
9.6.2	可屏蔽中断的硬件结构	264
9.6.3	硬件中断和软件中断的区别	266
9.7	日时钟中断	267
9.8	实地址模式定时中断程序设计	268
9.8.1	定时中断程序的设计方法	268
9.8.2	定时中断程序设计举例	269
9.9	实时时钟中断	277
9.9.1	实时时钟电路	277
9.9.2	周期中断	278
9.9.3	报警中断	281
9.10	键盘中断	285
9.10.1	键盘中断全过程	285
9.10.2	键代码生成	286
9.11	驻留程序	290
9.11.1	驻留程序的设计方法	290



9.11.2 驻留程序设计举例	293
9.11.3 驻留程序的解驻	296
习题	300
第 10 章 微型计算机系统串行通信	301
10.1 串行通信基础	301
10.1.1 串行通信类型	301
10.1.2 串行数据传输方式	302
10.1.3 串行异步通信协议	303
10.2 可编程串行异步通信接口芯片 8250	305
10.2.1 8250 的内部结构	306
10.2.2 8250 的引脚功能	307
10.2.3 8250 内部寄存器	310
10.2.4 8250 的初始化编程	314
10.3 串行通信程序设计	315
10.3.1 BIOS 通信软件	315
10.3.2 串行通信的外部环境	318
10.3.3 串行通信程序设计	319
10.4 可编程串行通信接口芯片 8251A	323
习题	333
第 11 章 并行 I/O 接口	335
11.1 可编程并行 I/O 接口芯片 8255A	335
11.1.1 8255A 的内部结构及外部引脚	335
11.1.2 8255A 的控制字与初始化编程	338
11.1.3 8255A 的工作方式	339
11.2 8255A 应用	347
11.3 打印机并行接口	356
11.3.1 打印机并行接口标准	356
11.3.2 打印机适配器	357
11.3.3 打印机接口编程	359
习题	364
第 12 章 DMA 控制器	365
12.1 概述	365
12.2 8237A DMA 控制器	366
12.2.1 8237A 的内部结构和引脚功能	366

12.2.2	8237A 内部寄存器	370
12.2.3	8237A 的时序	374
12.3	8237A 的应用	376
12.3.1	8237A 的初始化编程	376
12.3.2	8237A 在 IBM PC/AT 系统中的应用	377
习题	378
第 13 章	数模和模数转换	379
13.1	数模转换	379
13.1.1	数模转换原理	379
13.1.2	DAC 0832 简介	380
13.2	模数转换	382
13.2.1	模数转换原理	382
13.2.2	ADC 0809 简介	383
习题	385
第 14 章	保护模式及其编程	386
14.1	保护模式下的存储管理	386
14.1.1	分段管理	387
14.1.2	分页管理	391
14.1.3	虚拟存储器	393
14.1.4	保护机制	394
14.2	保护模式下的程序调用和转移	396
14.2.1	系统段描述符、门描述符和任务状态段	396
14.2.2	任务内的段间转移	400
14.2.3	任务间的转移	402
14.3	保护模式下的中断和异常	403
14.3.1	中断和异常分类	403
14.3.2	中断和异常类型	404
14.3.3	中断和异常的处理过程	405
14.3.4	中断和异常处理后的返回	406
14.4	保护模式下的输入/输出保护	407
14.5	操作系统类指令	408
14.5.1	实地址模式和任何特权级下可执行的指令	409
14.5.2	实地址模式和在特权级 0 下可执行的指令	409
14.5.3	只能在保护模式下执行的指令	410
14.6	保护模式下的程序设计	411

14.6.1	实地址模式与保护模式切换	411
14.6.2	保护模式下中断和异常程序设计	421
14.6.3	输入/输出保护及任务切换	429
习题	435
第 15 章	Windows 汇编语言编程初步	436
15.1	Windows 基础	436
15.2	Win32 汇编源程序的格式	438
15.2.1	源程序结构	438
15.2.2	Windows API 函数的应用	439
15.3	Win32 汇编可执行文件的生成	442
15.3.1	汇编和链接	443
15.3.2	调试 Win32 汇编程序	445
15.4	Win32 汇编基本语法	445
15.4.1	标号和变量	445
15.4.2	结构	447
15.4.3	子程序	448
15.4.4	高级语法	450
15.5	创建 Windows 下的窗口程序	454
15.5.1	窗口程序的运行过程	454
15.5.2	窗口程序示例	455
习题	461
参考文献	462

计算机基础

1.1 计算机中的数制

数制是数的表示方法。可以用各种进制来表示数,如二进制、十进制、八进制和十六进制等。由于使用电子器件表示两种状态比较容易实现,也便于存储和运算,所以,电子计算机中一般采用二进制数。但人们又习惯于使用十进制,因此在学习和掌握计算机的原理之前,需要了解各种进制的表示法及其相互关系和转换方法。

1.1.1 常用计数制

1. 十进制数

在程序设计中,广泛使用十进制数。十进制数的特点是:每一位有 0~9 这 10 种数码,故基数为 10,高位权是低位权的 10 倍,加减运算的法则为“逢十进一,借一当十”。

2. 二进制数

在电子计算机内部,所有信息都以二进制数形式出现。二进制数的特点是:只有两个不同的数字符号,即 0 和 1,因此基数为 2,高位权是低位权的 2 倍,加减运算的法则为“逢二进一,借一当二”。

3. 十六进制数

十六进制数是把 4 位二进制数作为一组,每一组用等值的十六进制数来表示。十六进制数的特点是:每一位有 0~9 和 A~F 这 16 种数码,因此基数为 16,高位权是低位权的 16 倍,加减运算的法则为“逢十六进一,借一当十六”。

4. 二-十进制数

二进制对计算机而言是最方便的,但是人们习惯于用十进制来表示数。为解决这一矛盾可采用二-十进制数。二-十进制数是计算机中十进制数的表示方法,就是用 4 位二进制数编码表示 1 位十进制数,简称为 BCD 码(Binary Coded Decimal)。

用四位二进制数编码表示一位十进制数,有多种表示方法,计算机中常用的是 8421

BCD 码,它的表示规则,以及与十进制之间的等价关系如表 1-1 所示。

表 1-1 BCD 码与十进制数的转换

二进制数	十进制数	BCD 码	二进制数	十进制数	BCD 码
0000	0	0000	1000	8	1000
0001	1	0001	1001	9	1001
0010	2	0010	1010	10	非法 BCD 码
0011	3	0011	1011	11	非法 BCD 码
0100	4	0100	1100	12	非法 BCD 码
0101	5	0101	1101	13	非法 BCD 码
0110	6	0110	1110	14	非法 BCD 码
0111	7	0111	1111	15	非法 BCD 码

例如: $(3456)_{10} = (0011\ 0100\ 0101\ 0110)_{\text{BCD}}$

BCD 码是十六进制数的一个子集,1010~1111 是非法 BCD 码。

1.1.2 数制转换

1. 二进制、十六进制数→十进制数

二进制、十六进制以至任意进制的数转换为十进制数的方法较简单,根据按权展开式把每个数位上的代码和该数位的权值相乘,再求累加和即可得到等值的十进制数。如:

$$(1101.11)_2 = 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = (13.75)_{10}$$

$$(\text{E5A})_{16} = 14 \times 16^2 + 5 \times 16^1 + 10 \times 16^0 = (3674)_{10}$$

2. 十进制数→二进制数

十进制数转换为二进制数时,根据该十进制数的类型来决定转换方法。

(1) 十进制整数→二进制数

方法为:“除 2 取余”,即十进制整数被 2 除,取其余数,商再被 2 除,取其余数……直到商为 0 时结束运算。然后把每次得到的余数按倒序规律排列,即可得到等值的二进制数。如:

$$N = (14)_{10} = (1110)_2$$

运算过程为: $14 \div 2 = 7$

余数 = 0 …… D_0

$$7 \div 2 = 3$$

余数 = 1 …… D_1

$$3 \div 2 = 1$$

余数 = 1 …… D_2

$$1 \div 2 = 0$$

余数 = 1 …… D_3

所以

$$N = D_3 D_2 D_1 D_0 = (1110)_2$$

(2) 十进制纯小数→二进制数

方法为:“乘 2 取整”,即把十进制纯小数乘以 2,取其整数(不参加后继运算),乘积的小数部分再乘以 2,取整……,直到乘积的小数部分为 0。然后把每次乘积的整数部分

按正序规律排列,即可得到等值的二进制数。如:

$$N=(0.8125)_{10}=(0.1101)_2$$

运算过程为: $0.8125 \times 2 = 1.625$	乘积的整数部分 = $1 \cdots \cdots D_{-1}$
$0.625 \times 2 = 1.25$	乘积的整数部分 = $1 \cdots \cdots D_{-2}$
$0.25 \times 2 = 0.5$	乘积的整数部分 = $0 \cdots \cdots D_{-3}$
$0.5 \times 2 = 1.0$	乘积的整数部分 = $1 \cdots \cdots D_{-4}$

所以 $N=(0.1101)_2$

有些纯小数,不断地“乘 2 取整”也不能使其乘积的小数部分为 0,此时只能进行有限次运算,根据需要取其近似值。

(3) 十进制带小数→二进制数

方法为:整数部分“除 2 取余”,小数部分“乘 2 取整”,然后再进行组合。例如:

$$(14.8125)_{10}=(1110.1101)_2$$

3. 二进制数→十六进制数

以小数点为界,四位二进制数为一组,不足四位用 0 补全,然后每组用等值的十六进制数表示。如:

$$(1101110.11)_2=(0110\ 1110.1100)_2=(6E.C)_{16}$$

在汇编语言中十六进制数用后缀“H”表示。所以:

$(1A2B)_{16}$ 应写成 1A2BH

4. 十六进制数→二进制数

把十六进制数的每一位用等值的二进制数来替换,如:

$$(17E.58)_{16}=(0001\ 0111\ 1110.0101\ 1000)_2=(101111110.01011)_2$$

1.2 计算机中数据的编码

计算机的中心任务就是处理信息。对于计算机系统,不同的信息应有不同的编码方法。计算机处理的信息,主要有数值数据和非数值数据两大类。数值数据是指日常生活中接触到的数字类数据,主要用来表示数量的多少,可以比较大小;非数值数据中最常用的数据是字符型数据,它可以用来表示文字信息,供人们直接阅读和理解;其他的非数值类型数据主要用来表示图画、声音和活动图像等。

1.2.1 数值数据的编码与运算

计算机中使用的数值数据,有无符号数和有符号数两种。在计算机中如何表示一个有符号数呢?最常用的方法是:把二进制数的最高一位定义为符号位,符号位为 0 表示正数,符号位为 1 表示负数,这样就把符号“数值化”了。有符号数的运算,其符号位上的 0 或 1 也被看作数值的一部分参加运算。

通常,把用“+”、“-”表示的数称为真值数,把用符号位上的 0、1 表示正、负的数称为机器数。机器数可以用不同的方法来表示,常用的有原码、反码和补码表示法。

1. 机器数的原码、反码和补码

数 X 的原码记作 $[X]_{\text{原}}$,反码记作 $[X]_{\text{反}}$,补码记作 $[X]_{\text{补}}$ 。

例如,当机器字长 $n=8$ 时:

	符号 ↓		符号位 ↓
设真值数	$X=+5=+0000101$	原码机器数写成	$[X]_{\text{原}}=00000101$
	$X=-5=-0000101$		$[X]_{\text{原}}=10000101$
	$X=+0=+0000000$		$[X]_{\text{原}}=00000000$
	$X=-0=-0000000$		$[X]_{\text{原}}=10000000$
设真值数	$X=+5=+0000101$	反码机器数写成	$[X]_{\text{反}}=00000101$
	$X=-5=-0000101$		$[X]_{\text{反}}=11111010$
	$X=+0=+0000000$		$[X]_{\text{反}}=00000000$
	$X=-0=-0000000$		$[X]_{\text{反}}=11111111$
设真值数	$X=+5=+0000101$	补码机器数写成	$[X]_{\text{补}}=00000101$
	$X=-5=-0000101$		$[X]_{\text{补}}=11111011$
	$X=+0=+0000000$		$[X]_{\text{补}}=00000000$

由上述例子可以得出以下结论:

- ① 机器数比真值数多一个符号位。
- ② 正数的原、反、补码的数值部分和真值数相同。
- ③ 负数原码的数值部分与真值相同;
负数反码的数值部分为真值数按位取反;
负数补码的数值部分为真值数按位取反末位加 1。
- ④ 没有负零的补码,或者说负零的补码和正零的补码相同。
- ⑤ 由于补码表示的机器数更适合运算,为此计算机系统中负数一律用补码表示。
- ⑥ 机器字长为 n 位的原码数,其真值范围是 $-(2^{n-1}-1) \sim +(2^{n-1}-1)$;
机器字长为 n 位的反码数,其真值范围是 $-(2^{n-1}-1) \sim +(2^{n-1}-1)$;
机器字长为 n 位的补码数,其真值范围是 $-(2^{n-1}) \sim +(2^{n-1}-1)$ 。

2. 整数补码的运算

为理解补码数是怎样进行加减运算的,首先引入几个概念。

(1) 模

模是计量器的最大容量。一个 4 位寄存器能够存放 0000~1111 共计 16 个数,因此它的模为 2^4 。一个 8 位寄存器能够存放 $0\cdots 0 \sim 1\cdots 1$,共计 256 个数,因此它的模为 2^8 ,以此类推,32 位寄存器的模是 2^{32} 。

(2) 有模的运算

凡是用器件进行的运算都是有模运算。运算之后,符号位向更高位的进位值无论是0还是1,都被运算器“丢弃”,而保存在“进位标志触发器”中。对于有符号数的运算,进位值不能统计在运算结果之中。而对于无符号数运算,其进位值则是运算结果的一部分,如进位值为1,表示运算结果已经超出了运算器所能表示的范围,仅用运算器的内容作为运算结果是不正确的。

(3) 求补运算

以下是一个由真值求补码的例子,机器字长 $n=8$:

设 $X=+75$,则 $[X]_{补}=01001011$ 。设 $X=-75$,则 $[X]_{补}=10110101$ 。

即:对 $[+X]_{补}$ 按位取反末位加1,就得到 $[-X]_{补}$ 。

对 $[-X]_{补}$ 按位取反末位加1,就得到 $[+X]_{补}$ 。

因此,“求补运算”就是指对一补码机器数进行“按位取反,末位加1”的操作。通过求补运算可以得到该数负真值的补码。

鉴于补码数具有这样的特征,用补码表示有符号数,则减法运算就可以用加法运算来替代,计算机中只需设置加法运算器就可以了。

(4) 整数补码的运算

采用补码进行加法运算的规则为:

$$[X+Y]_{补}=[X]_{补}+[Y]_{补}$$

其中, X 、 Y 为正负数皆可,符号位参加运算。

补码减法的规则是:

$$[X-Y]_{补}=[X]_{补}+[-Y]_{补}$$

其中, X 、 Y 为正负数皆可,符号位参加运算。

当真值满足下列条件时,应用上述规则就可得到正确的运算结果:

$$-2^{n-1} \leq (X,Y,X \pm Y) < 2^{n-1}$$

其中, n 为字长,运算以 2^n 为模。

【例 1.2.1】 设 $X=66$, $Y=51$,以 2^8 为模,补码运算求 $X \pm Y$ 。

解 因为 $[X]_{补}=01000010$, $[Y]_{补}=00110011$, $[-Y]_{补}=11001101$

$\begin{array}{r} [X]_{补} \quad 01000010 \\ +) [Y]_{补} \quad 00110011 \\ \hline [X+Y]_{补} \quad 01110101 \end{array}$	$\begin{array}{r} [X]_{补} \quad 01000010 \\ +) [-Y]_{补} \quad 11001101 \\ \hline [X-Y]_{补} \quad 1 \quad 00001111 \end{array}$
---	--

↓

被运算器丢失

所以 $X+Y=+117$ $X-Y=+15$

【例 1.2.2】 以 2^8 为模,补码运算求 $66+99,-66-99$ 。

解 因为 $[66]_{补}=01000010$, $[99]_{补}=01100011$

$[-66]_{补}=10111110$, $[-99]_{补}=10011101$

$\begin{array}{r} [66]_{\text{补}} \quad 01000010 \\ +) [99]_{\text{补}} \quad 01100011 \\ \hline [66+99]_{\text{补}} \quad 10100101 \end{array}$	$\begin{array}{r} [-66]_{\text{补}} \quad 10111110 \\ +) [-99]_{\text{补}} \quad 10011101 \\ \hline [-66-99]_{\text{补}} \quad 1 \quad 01011011 \end{array}$
	\downarrow 被运算器丢失

得到 $66+99=-91$ $-66-99=+91$

由上两例可以看出,不论被加数、加数是正数还是负数,只需直接用它们的补码(包括符号位)进行相加运算,当结果不超出补码表示范围时,运算结果是正确的补码;而当运算结果超出补码表示范围时,运算结果则是不正确的。运算结果超出了寄存器所能表示的范围,称为溢出。对于有符号数的加减运算,当参加运算的两个数的符号位相同而和运算结果的符号位相异时,则表示发生了溢出。

3. 无符号数

在处理某些问题时,若参与运算的数都是正数,如学生成绩、职工工资、字符编码和内存地址等。存放这些数时如保留符号位则没有实际意义。为了扩大寄存器所能表示的数的范围,可取消符号位。这样,一个数的最高位不再是符号位而是数值的一部分了,这样的数被称为“无符号数”。因此:

8 位字长的无符号数其数值范围是 $0 \sim 255$;

32 位字长的无符号数其数值范围是 $0 \sim 4294967295$ 。

计算机存储部件只知道它的内容是一串 0、1 代码。也就是说,只有程序员才能决定一个数的物理意义。

假设一个 32 位寄存器的内容是 $(11111111111111111111111111111111)_2$

若它是无符号数,其真值等于 4294967295。

若它是补码数,其真值等于 -1 。

若它是反码数,其真值等于 -0 。

注意:无符号数的加法运算,结果的进位位为 1 时,表示有溢出,进位值是和的一部分,不能随意丢弃。

1.2.2 字符的编码

在微型计算机系统中,键盘输入、打印输出和 CRT 显示的字符最常用的是美国信息交换标准代码,即 ASCII 码(American Standard Code for Information Interchange)。标准 ASCII 码用 7 位二进制数作为字符的编码,但由于计算机通常用 8 位二进制数代表一个字节,故标准 ASCII 码也写成 8 位二进制数,但最高位 D_7 位恒为 0。 $D_6 \sim D_0$ 代表字符的编码。表 1-2 为字符的 ASCII 码表。

表 1-2 标准 ASCII 码字符表

L \ H								
	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	.	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	\$	4	D	T	d	t
0101	ENG	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(8	H	X	h	x
1001	HT	EM)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[k	{
1100	FF	FS	,	<	L	\	l	
1101	CR	GS	-	=	M]	m	}
1110	SO	RS	.	>	N	↑	n	~
1111	SI	US	/	?	O	←	o	DEL

注：H 为高 3 位，L 为低 4 位。

NUL	空	DLE	数据键换码	SOH	标题开始
DC1	设备控制 1	STX	正文开始	DC2	设备控制 2
ETX	正文结束	DC3	设备控制 3	EOT	传输结束
DC4	设备控制 4	ENG	询问	NAK	否定
ACK	认可	SYN	同步字符	BEL	报警(可听见声音)
ETB	信息组传送结束	BS	退一格	CAN	作废
HT	横向制表	EM	纸尽	LF	换行
SUB	减	VT	纵向制表	ESC	换码
FF	走纸控制	FS	文字分隔符	CR	回车
GS	组分分隔符	SO	移位输出	RS	记录分隔符
SI	移位输入	US	单元分隔符	SP	空格
DEL	删除				

1.3 浮点数基本概念

人们常用的数据一般有三种：纯整数(如二进制数 1101)、纯小数(如二进制数 0.1101)和既含整数又含小数的数(如二进制数 1.1101)。在计算机中,表示这三种数有两种方法：定点表示法和浮点表示法。计算机中数的小数点位置固定的表示法称为定点表示法,用定点表示法表示的数称为定点数。计算机中数的小数点位置不固定的表示法称为浮点表示法,用浮点表示法表示的数称为浮点数。值得注意的是：小数点在计算机中是不表示出来的,而是隐含在用户规定的位置上。一般地,纯整数和纯小数用定点表示法比较方便;而既含整数又含小数的数用浮点表示法时比较实用且便于运算。

1.3.1 浮点数

一个二进制带小数可以写成许多种等价形式,例如:

$$\pm 101101.0101 = \pm 1.011010101 \times 2^{+5}$$

$$\pm 101101.0101 = \pm 0.1011010101 \times 2^{+6}$$

$$\pm 101101.0101 = \pm 0.01011010101 \times 2^{+7}$$

$$\pm 101101.0101 = \pm 1011010101 \times 2^{-4}$$

任何一个二进制数 N (含整数和小数两个部分)都可写成统一的格式:

$$N = \pm S \times 2^{\pm J}$$

$\downarrow \downarrow$
尾尾
符数

$\downarrow \downarrow$
阶阶
符码

可以得出结论:

① 用阶码和尾数两部分共同表示一个数,这种表示方法称为数的浮点表示法。

② 阶码和阶符的物理意义:阶码表示小数点的实际位置。

如: $0.01011010101 \times 2^{+7}$

表达式中阶符和阶码为+7,表示把尾数的小数点向右移动7位就是小数点的实际位置,因此该数等于:

101101.0101

如: 1011010101×2^{-4}

表达式中阶符和阶码为-4,表示把尾数的小数点向左移动4位就是小数点的实际位置,因此该数等于:

101101.0101

③ 规格化的浮点真值数。

二进制带小数,可以写成若干种等价的形式,其中只有一种被称为“规格化的浮点数”。

规格化的浮点真值数满足以下两个条件:

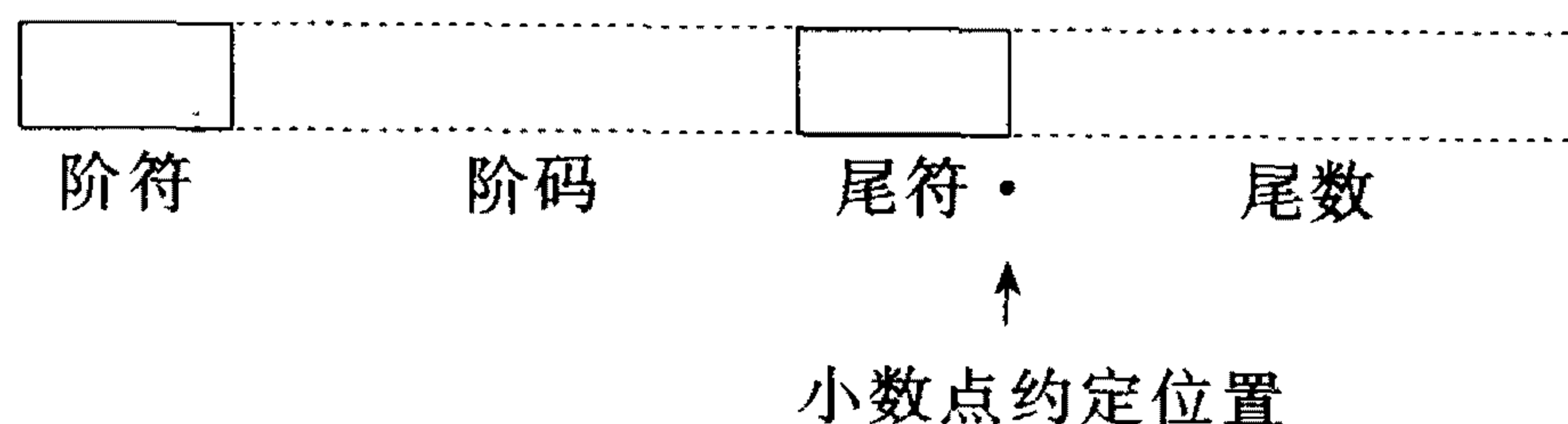
尾数为纯小数,且小数点后面是1不是0;

阶码为整数(正整数或者负整数)。

因此,上述的二进制带小数只有 $\pm 0.1011010101 \times 2^{+6}$ 是规格化的浮点真值数。

1.3.2 浮点机器数

计算机硬件存储一个浮点数的常用格式如下:



- 在一个字长(8 位、16 位或 32 位二进制数)中：
- 选用 1 位存放阶符,阶符为 0 表示阶码为正数,阶符为 1 表示阶码为负数；
- 选用若干位存放阶码,阶码为整数；
- 选用 1 位存放尾符(数符),尾符为 0 表示尾数为正数,尾符为 1 表示尾数为负数；
- 选用若干位存放尾数,尾数为纯小数；
- 尾符和尾数之间是小数点的约定位置。

由于浮点数由阶码和尾数两部分组成,这两部分都是有符号的。它们用什么码制表示呢？归纳起来浮点机器数有两种：

- ① 阶码和尾数采用相同的码制；
- ② 阶码和尾数采用不同的码制。

【例 1.3.1】 设字长为 16 位,其中：阶符 1 位,阶码 4 位,尾符 1 位,尾数 10 位。要求把 $X=-101101.0101$ 写成规格化的浮点补码数,阶码和尾数均用补码表示。

解 首先把 X 写成规格化的浮点真值数 $X=-0.1011010101\times 2^{+6}$,
则规格化的浮点补码数如下：

0	0110	1	0100101011
阶符	阶码	尾符	尾数

【例 1.3.2】 设阶码用原码表示,尾数用补码表示,求下列浮点机器数的真值。

1	0010	1	0010011001
阶符	阶码	尾符	尾数

解 真值 $=-0.1101100111\times 2^{-2}$

1.3.3 浮点数的数值范围

“数值范围”是指机器数所能表示的真值范围。在定字长条件下,浮点数所能表示的真值范围比定点数大,而且分配给阶码的位数越多,所能表示的数的范围越大,但是由于尾数的位数相应减少,所以数的精度减小。

【例 1.3.3】 设字长为 16 位,其中：阶符 1 位,阶码 5 位,尾符 1 位,尾数 9 位,当阶码和尾数均用补码表示时,数值范围是多大？当阶码和尾数都用原码表示时,数值范围是多大？

解 图 1-1 给出了两种情况下的数值范围。

当阶码占 M 位,尾数占 N 位,而且阶码和尾数采用不同码制表示的时候,其数值范围又是多大？读者从上例能得到启发。

需要说明的是：本书涉及的微型计算机中指令运算的操作数是定点整数,即用汇编语言编程涉及的都是整数,对于浮点数,本书不再详述。

1	1 1 1 1 1	0	1 1 1 1 1 1 1 1 1
---	-----------	---	-------------------

阶码正最大(补码)

尾数正最大(补码)

真值最大 = $+(1-2^{-9}) \times 2^J$

0	1 1 1 1 1	1	0 0 0 0 0 0 0 0 0
---	-----------	---	-------------------

阶码正最大(补码)

尾数负最大(补码)

真值最小 = -1×2^J 所以,数值范围 = $-1 \times 2^J \sim +(1-2^{-9}) \times 2^J$ 式中 $J = 2^5 - 1$

0	1 1 1 1 1	0	1 1 1 1 1 1 1 1 1
---	-----------	---	-------------------

阶码正最大(原码)

尾数正最大(原码)

真值最大 = $+(1-2^{-9}) \times 2^J$

0	1 1 1 1 1	1	1 1 1 1 1 1 1 1 1
---	-----------	---	-------------------

阶码正最大(原码)

尾数负最大(原码)

真值最小 = $-(1-2^{-9}) \times 2^J$ 所以,数值范围 = $-(1-2^{-9}) \times 2^J \sim +(1-2^{-9}) \times 2^J$ 式中 $J = 2^5 - 1$

图 1-1 浮点数的数值范围

1.4 计算机系统的基本组成

计算机系统由“硬件”和“软件”两大部分组成,硬件是构成计算机的设备实体,软件是指为了运行、管理和维修计算机而编制的各种程序。

1.4.1 计算机系统的硬件组成

现代计算机的硬件结构仍然是在冯·诺依曼提出的计算机逻辑结构和存储程序概念的基础上建立起来的。“存储程序”就是指将指令、数据以二进制形式存入计算机系统的存储器中。“程序控制”就是指计算机启动后,自动取出并执行存于存储器中的指令,完成预定的操作。

基于这种思想,计算机的硬件系统基本上由运算器、控制器、存储器、输入/输出接口和输入/输出设备、电源系统等组成,如图 1-2 所示。

其中,运算器和控制器合称为中央处理器(Central Processing Unit, CPU)。CPU 与存储器系统、I/O 接口、电源系统等组成了计算机系统的“主机”,输入/输出设备被称为外部设备。

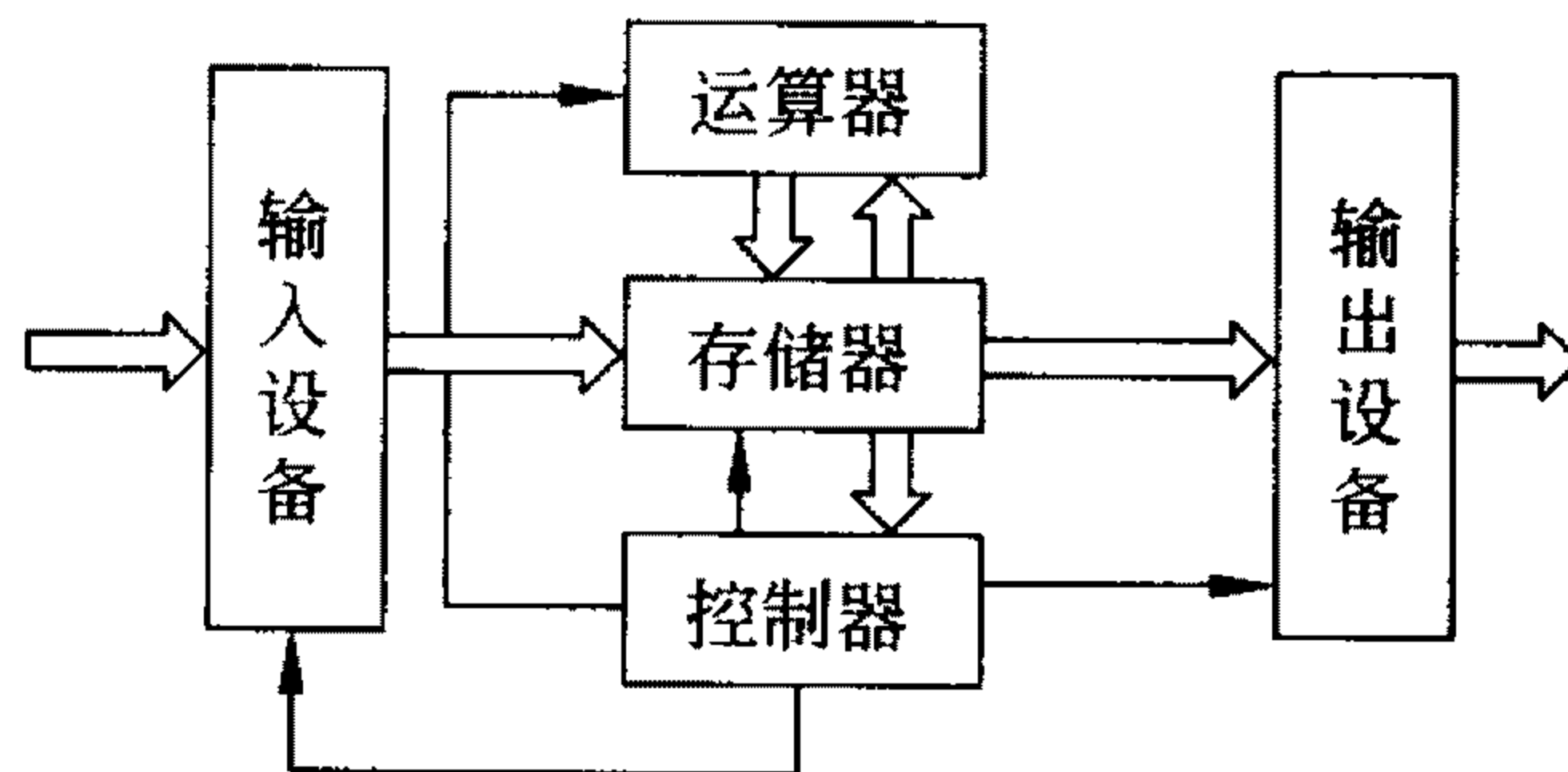


图 1-2 计算机系统的硬件组成

1. 存储器

这里所讲的存储器是指计算机系统的内存储器,也称主存储器,简称为内存。硬盘、软盘被称为外存储器,即辅助存储器。存储器用来存放指令和数据。

2. 运算器

运算器(Arithmetic Logic Unit, ALU)是进行算术运算和逻辑运算的部件,它也是指令的执行部件。

3. 控制器

控制器是计算机的指挥中心。它负责对指令进行译码,产生出整个指令系统所需要的全部操作的控制信号,控制运算器、存储器和输入/输出接口等部件完成指令规定的操作。

4. 输入设备

输入设备通过输入接口电路将程序和数据输入内存。最常见的输入设备是键盘和鼠标。

5. 输出设备

CPU 通过输出接口电路将程序运行的结果、程序和数据送到输出设备上。最常见的输出设备有显示器和打印机。

1.4.2 计算机系统的软件组成

计算机如不配置软件,被称为“裸机”,仅有裸机是不能做任何事情的,必须得有软件的配合。计算机的软件又分为系统软件和应用软件两大类。

1. 系统软件

它是使用、管理计算机本身的软件。例如:

- ① 面向计算机管理的软件,如操作系统等;
- ② 数据库管理系统: SYBASE、FoxPro 等;
- ③ 计算机网络管理软件;
- ④ 语言处理、服务性软件。

2. 应用软件

它是计算机用户在各自的业务领域中开发和使用的各种软件,是为解决某一实际问题而编制的程序。例如:天气预报中的数据处理、建筑业中的框架设计、企业的财务管理、工厂的仓库管理、学校的辅助教学等。编辑程序、宏汇编程序、链接程序是汇编语言程序设计使用的 3 个应用软件。

1.4.3 微型计算机的硬件结构

电子计算机通常按体积、性能和价格分为巨型机、大型机、中型机、小型机和微型机五类。微型计算机的系统结构和基本工作原理与其他几类计算机并没有本质上的区别,所不同的是微型机广泛采用了集成度相当高的器件和部件。微型计算机硬件的核心是微处理器(Micro-processing Unit, MPU),一般就称为 CPU。CPU 集成了运算器、控制器、寄存器组和存储管理等部件。微型机简单的结构示意图如图 1-3 所示。以微型计算机为主体,配上系统软件和外设之后,就构成了微型计算机系统。

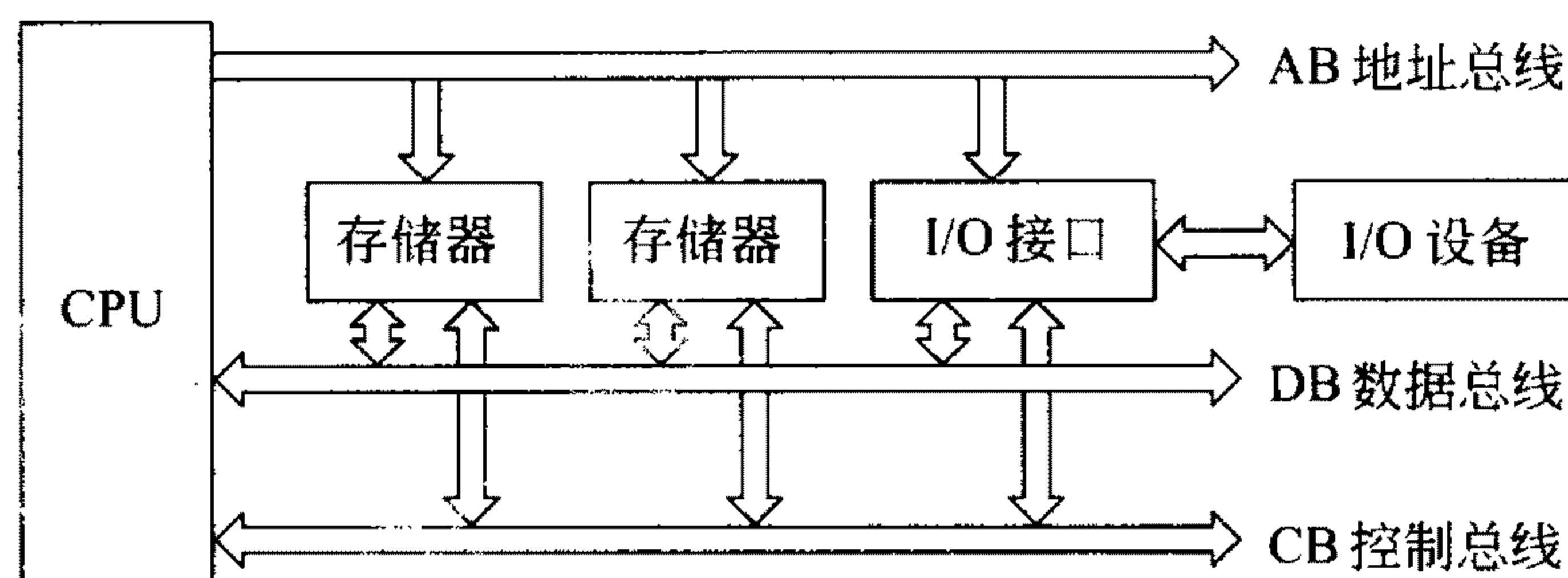


图 1-3 微型机硬件结构示意图

1. 总线

总线是连接 CPU 与存储器、I/O 接口的公共导线,是各部件信息传输的公共通道。微型计算机系统有 3 条总线(3“条”是习惯说法,实际上每一条总线都有若干根),它们是地址总线(Address Bus)、数据总线(Data Bus)和控制总线(Control Bus)。

地址总线传输地址信息,用来寻址存储单元和 I/O 端口。地址总线的“宽度”决定了系统内存的最大容量。8088/8086 有 20 根地址线,只能寻址 1 兆字节内存,80286 有 24 根地址线,可寻址 16 兆字节内存,80486 有 32 根地址线,可寻址 4G 字节的存储单元。

数据总线传输数据信息,8088 CPU 内部各模块之间的数据线是 16 位,但 CPU 与存储器、I/O 端口之间传递信息使用 8 位数据线,所以称 8088 是准 16 位微处理器;80286 内/外数据线都是 16 位宽度,是 16 位微处理器;80486 有 32 根数据线,这意味着 CPU 和存储器、I/O 端口每一次可以传输 4 个字节的数据。

控制总线对于不同的 CPU 来讲,其条数不一样。控制线向系统各部件发出(或接收)各种控制信号。

从信息流向的角度讲,地址总线通常是单向总线,地址信息由 CPU 发出。数据总线是双向总线。控制总线也是双向总线,其中大部分控制线是单向的,它们是 CPU 发出的操作命令,或者是其他部件向 CPU 提出的请求信号,只有少数控制线是双向控制线。

2. 存储器

微型计算机中的存储器主要有 ROM 和 RAM 两种。

ROM 是只读存储器。在微型计算机系统中,ROM 存放基本输入/输出系统程序(Basic Input/Output System, BIOS),BIOS 是微型计算机系统最底层的系统管理程序。

RAM 为随机(读/写)存储器,即通常所讲的内存。随机存储器大多选用动态 RAM,目前的微型计算机中也部分采用静态 RAM 构成高速缓冲存储器 Cache。

3. I/O 接口

顾名思义,I/O 接口是 CPU 与 I/O 设备之间交换信息的接口电路。接口电路中有暂存数据的寄存器,为了便于 CPU 执行指令与之交换信息,系统也给这些寄存器编排地址,这些地址称为“端口地址”,接口电路中能与 CPU 交换信息的那些寄存器就称为“端口寄存器”。

1.4.4 微型计算机的分类和发展

1. 微型计算机的分类

微型计算机的分类方法很多。按微处理器的位数,可分为 1 位、4 位、8 位、16 位、32 位和 64 位机等。按功能和结构可分为单片机和多片机。按组装方式可分为单板机和多板机等。

利用大规模集成电路工艺将微型计算机的三大组成部分——CPU、内存和 I/O 接口集成在一片硅片上,这就是单片机(Single-Chip Computer)。使用专用开发装置就可以对它进行在线开发。单片机在工业过程控制、智能化仪表和家用电器中得到广泛的应用。若将微型计算机的 CPU、内存和 I/O 接口电路安装在一块印刷电路板上就组成了单板机,单板机结构简单,价格低廉,性能较好,经过开发以后,可用于过程控制、各种仪器仪表、机器的单机控制和数据处理等。微型计算机是多板机,它由主板及插在主板上的多个电路板组成。

2. 微型计算机的发展

由于微型计算机具有体积小、重量轻、价格低廉、可靠性高、结构灵活和应用面广的特点,它的发展速度大大超过了其他的机型。微型计算机的发展很大程度上取决于微处理器的发展,所以,微处理器的发展史就是微型计算机的发展史。自从 20 世纪 70 年代初第一个微处理器诞生以来,微处理器的性能和集成度几乎每两年提高一倍,而价格却降低了一个数量级。

微处理器的性能指标最主要的是字长和主频两个方面。字长就是指 CPU 能同时处理的数据位数,也称为数据宽度。字长越长,计算机的能力就越高,速度也越快,但集成度要求也越高,工艺越复杂。主频即 CPU 的时钟频率,这和 CPU 的运算速度密切相关,主频越高,运算速度也越快。

第一个微处理器是 1971 年美国 Intel 公司生产的 4004。它本来是为高级袖珍计算器设计的,但生产出来后,取得了意外的成功。于是,Intel 公司对它作了改进,正式生产了通用的 4 位微处理器 4040。Intel4040 以它的体积小、价格低等特点引起了许多部门和机构的兴趣。1972 年,Intel 公司又生产了 8 位的微处理器 8008。通常,人们将

Intel4004、4040 和 8008 称为第一代微处理器。这些微处理器的字长为 4 位或 8 位,集成度大约为 2000 管/片,时钟频率为 1MHz,平均指令执行时间约为 $20\mu\text{s}$ 。

后来出现了许多生产微处理器的厂家,1973 到 1977 年之间,这些厂家生产了多种型号的微处理器,其中设计最成功、应用最广泛的是 Intel 公司的 8080/8085、Zilog 公司的 Z80、Motorola 公司的 6800/6802 和 Rockwell 公司的 6502。人们通常把它们称为第二代微处理器。这些微处理器的时钟频率为 $2\sim 4\text{MHz}$,平均指令执行时间为 $1\sim 2\mu\text{s}$,集成度超过 5 000 管/片,其中的 8085、Z80 和 6802 的集成度都高达 10 000 管/片。在这个时期,微处理器的设计和生产技术已经相当成熟,配套的各类器件也很齐全。随后,微处理器在提高集成度、提高功能和速度以及增加外围电路的功能和种类方面得到了很大的发展。

1977 年左右,超大规模集成电路工艺已经成熟,一个硅片上可以容纳几万个管子。于是在 1978 到 1979 年间,一些厂家推出了性能可与过去中档小型计算机相比的 16 位微处理器,其中有代表性的三种芯片就是 Intel 公司的 8086/8088、Zilog 公司的 Z8000 和 Motorola 公司的 M68000。这些微处理器的时钟频率为 $4\sim 8\text{MHz}$,平均指令执行时间为 $0.5\mu\text{s}$,集成度为 20 000 \sim 60 000 管/片。人们将这一代微处理器称为超大规模集成电路微处理器。

1980 年以后,半导体厂家继续在提高电路的集成度、速度和功能方面取得了很大进展,相继推出了 Intel 80286、Motorola 68010 这样一些集成度高达 100 000 管/片、时钟频率为 10MHz、平均指令执行时间为 $0.2\mu\text{s}$ 的 16 位高性能微处理器。

1983 年以后,Intel 80386 和 Motorola 68020 相继推出。这两者都是 32 位的微处理器,时钟频率达 $16\sim 20\text{MHz}$,平均指令执行时间为 $0.1\mu\text{s}$,集成度高达 150 000 \sim 500 000 管/片。

在 1989 到 1995 年间,Intel 公司又相继推出了 80486 和 Pentium 高性能 32 位微处理器,其中的 Pentium 的集成度高达 3 100 000 管/片,时钟频率高达 150MHz。

1995 年 11 月至 2000 年 11 月期间,Intel 公司又陆续推出了 Pentium Pro、Pentium MMX、Pentium II、Pentium III 和 Pentium 4 微处理器,这些微处理器的内部都是 32 位数据宽度,所以都属于 32 位微处理器。在此过程中,微处理器的集成度和主频不断提高,目前市场上的 Pentium 4 微处理器集成了 4 200 万个晶体管,其时钟频率高达 2.4GHz,而主频超过 3GHz 的 Pentium 4 和 64 位微处理器 Itanium(安腾)也已推向市场。

在微型计算机的发展过程中,许多公司的微处理器的发展都采用了向下兼容的策略,每一种新的微处理器都对原有的系列产品保持兼容,从而使此前的软件都能够继续运行。Intel 公司的 x86 系列微处理器就是一个很好的例子。如 16 位微处理器 8086 指令系统,一方面兼容了低档微处理器的全部指令,另一方面被 32 位微处理器兼容。32 位微处理器 80386/80486/Pentium 的指令系统正是在 8086 基础上扩展和补充而成的。

习 题

1. 数制和码制转换

(1) $(11101.1011)_2 = (\quad)_{10}$

(2) $(147)_{10} = (\quad)_2 = (\quad)_{16}$

(3) $(3AC)_{16} = (\quad)_{10}$

(4) $(10010110)_{BCD} = (\quad)_2$

(5) 字长=8, $[-1]_{补} = (\quad)_{16}$

$[X]_{补} = (A5)_{16}$, 则 $X = (\quad)_{16}$

(6) 设字长=8 位, $X = (8E)_{16}$, 当 X 分别为原码、补码、反码和无符号数的时候, 其真值= $(\quad)_{16}$

(7) 字长=8, 用补码形式完成下列十进制数运算。要求有运算过程并讨论结果是否有溢出?

(a) $(+75) + (-6)$

(b) $(-35) + (-75)$

(c) $(-85) - (-15)$

(d) $(+120) + (+18)$

2. 设字长=16 位, 阶符 1 位, 阶码占 M 位, 尾符 1 位, 尾数 N 位。按下列要求写出浮点数的真值范围。

(1) 阶码用补码表示, 尾数用原码表示。

(2) 阶码用原码表示, 尾数用补码表示。

(3) 阶码、尾数均用原码表示。

(4) 阶码、尾数均用补码表示。

3. 微型计算机系统由哪几部分组成? 微处理器、微型计算机和微型计算机系统的关系是什么?

4. 微型计算机的 CPU、存储器和 I/O 接口通过哪三大总线互连在一起, 各自的功能是什么?

5. 什么是定点数、浮点数? 什么是无符号数、有符号数?

6. 写出下列数表示的无符号数和有符号数范围。

(1) 8 位二进制

(2) 16 位二进制

80x86 微处理器

微处理器是微型计算机系统的核心部件,是采用大规模或超大规模集成电路技术做成的半导体芯片。计算机系统中的各部件都是在微处理器的统一调度之下协调工作,所以它又被称为中央处理单元(Central Processing Unit,CPU)。

本章着重介绍 32 位微处理器的内部结构、对外接口信号、工作模式和典型的总线操作时序,以便读者可以了解微型计算机的工作原理。

2.1 Intel 微处理器发展简况

Intel 公司于 1981 年推出 8086 与 8088 微处理器,著名的 IBM XT 型计算机就是基于 8088 微处理器的。这两种 16 位的微处理器比以往的 8 位机功能更强大,地址线有 20 条,内存寻址范围为 1MB。它们的区别在于,8086 外部的数据也是 16 位,而 8088 的外部数据为 8 位。

1982 年,Intel 推出了 80286 芯片,该芯片含有 13.4 万个晶体管,80286 也是 16 位处理器,其频率比 8086 更高,它有 24 条地址线,内存寻址范围是 16MB。

80386 属于 32 位微处理器,其内部和外部数据总线都是 32 位,地址总线也是 32 位,可寻址 4GB 内存。它除了具有实地址模式和保护模式外,还增加了一种叫虚拟 86 的工作方式,可以通过同时模拟多个 8086 处理器来提供多任务能力。它有以下几种型号:80386SX,它是准 32 位处理器,数据总线是 16 位,其内部 32 位寄存器必须分两个 16 位的总线来读取,是 286 计算机与 386DX 计算机之间的过渡产品。80386DX 是真正的 32 位处理器,80386DX 的数据总线和内部寄存器都是 32 位。80386DX 还可以配上 80387 数字协处理器,以提高计算速度。386 处理器的主频有 16MHz、20MHz、25MHz、33MHz 和 40MHz 五种。除 Intel 公司生产 386 芯片外,还有 AMD,Cyrix,Ti 和 IBM 等公司也生产 386 芯片。

80486 简称 486,于 1989 年由 Intel 公司首先推出,集成了 120 万个晶体管。其时钟频率从 25MHz 逐步提高到 33MHz、50MHz。它也属于 32 位处理器。80486 是将 80386 和数字协处理器 80387 以及一个 8KB 的高速缓存集成在一个芯片内,并且在 80x86 系列中首次采用了 RISC 技术,可以在一个时钟周期内执行一条指令。它还采用了突发总线方式,大大提高了 CPU 与内存的数据交换速度。

Pentium(奔腾)是 Intel 公司于 1993 年推出的新一代微处理器,它集成了 310 万个晶体管。Pentium 微处理器使用更高的时钟频率,最初为 60MHz 和 66MHz,后提高到 200MHz,使用 64 位数据总线和 16KB 的高速缓存。奔腾 CPU 的出现进一步加快了 CPU 的更新速度,使 CPU 厂商竞争愈加激烈。Intel 公司为了防止别的公司侵权,就为新的 CPU 取名“Pentium”,而没有继续叫做 80586。接着 Intel 推出使用 MMX 技术的 Pentium MMX 的多能奔腾。它增加了 57 条多媒体指令,内部高速缓存增加到 32KB。最高频率是 233MHz。MMX 是 Multimedia Extension 的缩写,意即多媒体扩展,是一种基于多媒体计算以及通信功能的技术,它能生成高质量的图像、视频和音频,加速对声音图像的处理。Pentium Pro,中文称为高能奔腾,也称为 P6。它在 Pentium MMX 之前面市,使用大量新技术,还包含了 256KB 或 512KB 的高速缓存,主要应用在服务器上。

1997 年,个人计算机微处理器的领先者是 Intel 的 Pentium II、Pentium III。P II/P III 芯片内部集成 32KB 的高速缓存和 512KB 的二级缓存,使用了 MMX 和 AGP 技术。为了占有市场,采用新的封装结构,并采用了 SLOT 1 插槽与主板结合。Pentium III 就是大家以前关注很久的 Katmai,它采用了与 Pentium II 相同的 SLOT1 结构,具有 100MHz 的外频,其内部集成了 64KB 的一级缓存,512KB 的二级缓存仍然安装在 SLOT1 的卡盒内,工作频率是 CPU 的一半。不过仍提供了比 Pentium II 更强劲的性能,这主要表现在其新增加了 KNI 指令集。KNI 指令集中提供了 70 条全新的指令,可以大大提高 3D 运算、动画片、影像、音效等功能,增强了视频处理和语音识别的功能。这套指令集主要为浏览 WWW 网页而设计的。后来又推出了主频为 450MHz、500MHz 的型号。P III 可以安装在 P II 的 SLOT1 的主板上,不过要更新 BIOS 的内容,才能支持新增的 KNI 指令。

2000 年 6 月 Intel 推出了新型体系结构的 32 位微处理器芯片 Pentium 4,其起始主频为 1.2~1.5GHz,目前用于 PC 机的 Pentium 4 的主频已超过 3GHz。它增加了 144 条 SSE2 指令,采用了一系列的新技术来面向网络功能和图像功能,有超级流水线技术、跟踪性指令 Cache 技术和双沿指令快速执行机制等。

2000 年 11 月,Intel 公司又推出了第一代 64 位微处理器芯片 Itanium,标志着 Intel 的微处理器芯片进入了 64 位时代。其内部集成了 2.2 亿个晶体管,集成度几乎是 Pentium 的 10 倍,主要面向高档服务器和 workstation。它在 Pentium 的基础上又引入了多种新技术,如拥有三级 Cache,有 4 个整数执行部件 ALU 和 4 个浮点执行部件 FMAC 及 9 个功能通道,还有数量众多的寄存器,采用完全并行指令计算 EPIC(Explicitly Parallel Instruction Computing)技术使处理器具有更好的指令级并行能力,采用新机制的分支预测技术等。

2.2 32 位微处理器内部结构

Intel 8086 与随后推出的 8088 CPU 的内部运算、数据寄存和操作都比较类似,按 16 位进行,只是前者对外数据线是 16 位后者是 8 位而已。它们的内部被设计成独立的两个功能模块:总线接口单元 BIU(Bus Interface Unit)和执行单元 EU(Execution Unit),使取指令和执行指令可以并行操作,提高了总线的利用率。而后来推出的 80286,它的内

部结构除了具备 8086/8088 最基本功能外还增加了虚拟存储、特权保护、任务管理等功能,所以支持多用户和多任务系统。

20 世纪 80 年代中期开始,微处理器进入了 32 位时代。Intel 公司陆续推出的 32 位微处理器 80386、80486、Pentium、Pentium Pro 和 Pentium MMX 等一系列高性能微处理器,主要特点是将浮点运算部件集成在片内,并普遍采用了时钟倍频技术、流水线和并行及推测执行技术、虚拟存储及片内存储体分段分页双重管理和保护等技术,为在微型计算机环境下实现多用户多任务操作提供了有力的支持。

2.2.1 Pentium 内部结构

Pentium 微处理器的内部寄存器长度都为 32 位,但外部数据总线不像 386 和 486 那样是 32 位,而是 64 位,总线传输速度高达 66MHz。同时它具有 32 位地址总线,可直接寻址 4GB 的物理内存空间。它有两条相对独立的指令并行流水线,即 U 线和 V 线,内含高性能浮点处理部件及多媒体处理 MMX 部件,允许使用双精度浮点数实现由高速向量生成图形显示。Pentium 微处理器的内部结构如图 2-1 所示。

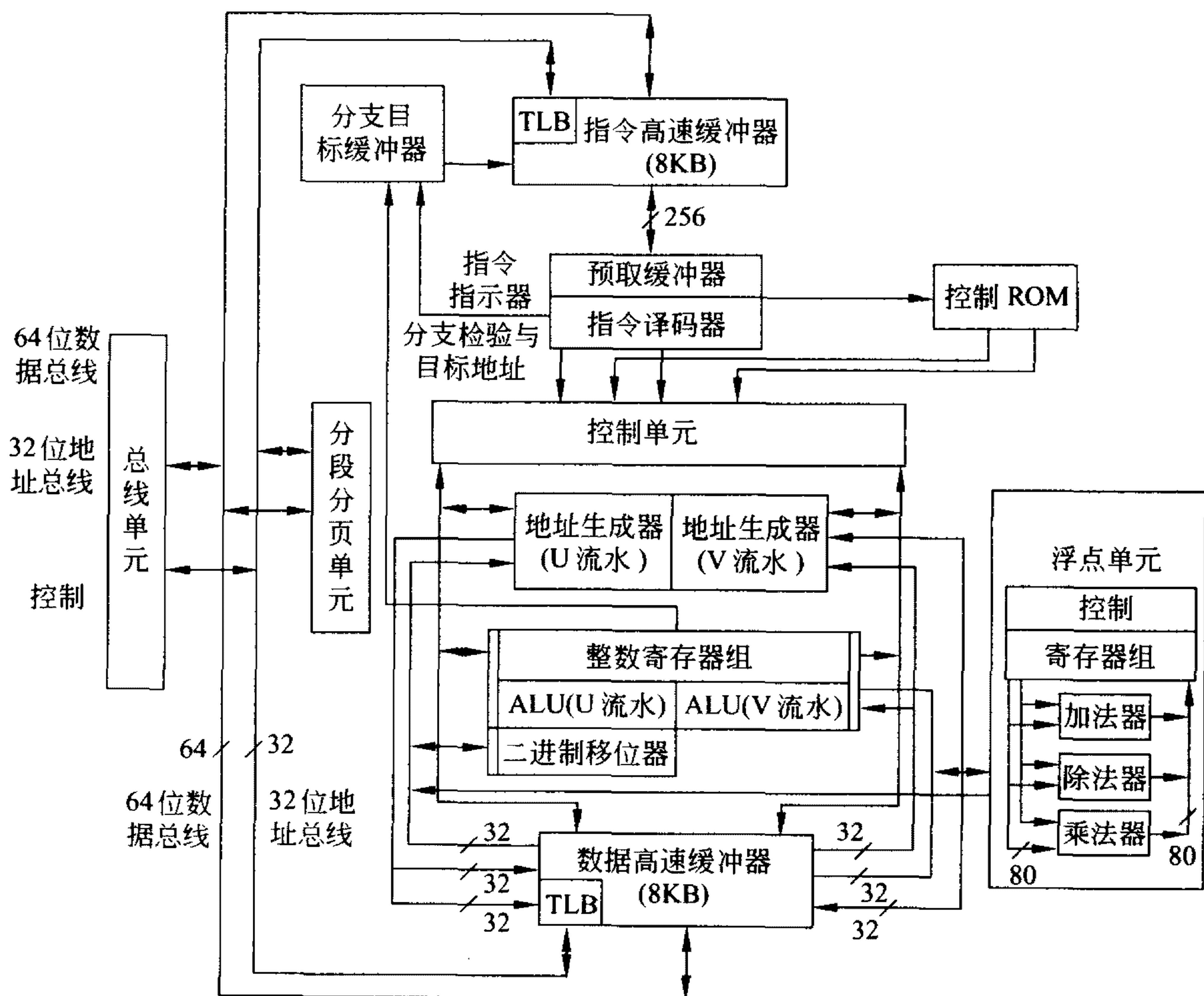


图 2-1 Pentium 微处理器的结构框图

从图 2-1 可以看出,Pentium 微处理器内部的主要部件有 10 个,分别是:总线接口部件、分段分页部件、U 流水线和 V 流水线、指令 Cache 和数据 Cache、指令预取部件、指令译码器、浮点处理部件 FPU、分支目标缓冲器 BTB、控制 ROM 和寄存器组。

总线接口部件实现 CPU 与系统总线的连接,其中包括 64 位数据线、32 位地址线和众多控制信号线,以此实现相互之间的信息交换,并产生相应的总线周期信号。

分段分页部件则实现将各种地址映射到内存物理地址的功能。

高速缓存即 Cache 是容量较小,速度很高的,可读写的 RAM,用来存放 CPU 最近要使用的数据和指令,Cache 可以加快 CPU 存取数据的速度,减轻总线负担。Cache 中的数据其实是主存中一小部分数据的复制品,所以,要时刻保持两者的相同,即保持数据一致性。在 Pentium 微处理器内部,指令 Cache 和数据 Cache 是分开的,目的是提高访问的命中率。

指令预取部件每次可以取两条指令,如果是简单指令,并且后一条指令不依赖前一条指令的执行结果,那么,指令预取部件便将两条指令分别送到 U 流水线和 V 流水线独立执行。

指令 Cache、指令预取部件将原始指令送到指令译码器,而分支目标缓冲器则在遇到分支转移指令时用来预测转移是否发生。

浮点处理部件 FPU 主要用于浮点运算,内含专用的加法器、乘法器和除法器,加法器和乘法器均能在 3 个时钟周期内完成相应的运算,除法器则在每个时钟周期内产生两位二进制商。

控制 ROM 中,含有 Pentium 微处理器的微代码,控制部件则直接控制流水线的操作。

2.2.2 Pentium 微处理器结构特点

80486 微处理器内部有 8 个基本模块,分别是:总线接口部件、指令预取部件、指令译码部件、执行部件、控制部件、存储管理部件、高速缓冲存储部件(Cache)和高性能浮点处理部件。由图 2-1 知,Pentium 微处理器内部有 10 个基本模块,和 80486 相比具有如下特点:

① Pentium 由“U”和“V”两条指令流水线构成了超标量流水线结构。每条流水线都有自己的 ALU、指令译码、地址生成、指令执行和回写五个步骤。其中 U 流水线可执行所有的整数和浮点指令,而 V 流水线中只能执行简单的整数指令和一条异常的 FXCH 指令。当一条指令完成预取步骤时,流水线就可以开始对另一条指令的操作,极大地提高了指令的执行速度。

② 重新设计的浮点部件。Pentium 的浮点部件在 80486 的基础上做了重新设计,其执行过程分为八级流水,使每个时钟周期能完成一个浮点操作。采用快速算法可使诸如 ADD、MUL 和 LOAD 等指令的运算速度提高至少三倍,在许多应用程序中利用指令调度和重叠(流水线)执行可使性能提高五倍以上。同时,对电路进行固化,用硬件来实现。

③ 独立的指令 Cache 和数据 Cache。Pentium 片内有两个 8KB 的 Cache——双路 Cache 结构,一个是指令 Cache,一个是数据 Cache。转换后备缓冲器(Translation Look-aside Buffer, TLB)的作用是将线性地址转换为物理地址。这两种 Cache 采用 $32 \times 8\text{bit}$ 线宽,是对 Pentium 的 64 位总线的有力支持。指令和数据分别使用不同的 Cache,使 Pentium 中数据和指令的存取减少了冲突,提高了性能。

Pentium 的数据 Cache 有两个接口,分别与 U 和 V 两条流水线相连,以便能在相同时刻向两个独立工作的流水线进行数据交换。当向已被占满的数据 Cache 中写数据时,将移走当前使用频率最低的数据,同时将其写回内存,这种技术称为 Cache 回写技术。由于 CPU 向 Cache 写数据和将 Cache 释放的数据写回内存是同时进行的,所以采用 Cache 回写技术将节省处理时间。

④ 分支预测。Pentium 提供了一个称为分支目标缓冲器(Branch Target Buffer, BTB)的小 Cache 来动态地预测程序的分支操作。当某条指令导致程序分支时,BTB 记下该条指令和分支目标的地址,并用这些信息预测该条指令再次产生分支时的路径,预先从该处预取,保证流水线的指令预取步骤不会空置。这一机构的设置,可以减少在循环操作时对循环条件的判断所占用的 CPU 时间。

⑤ 采用 64 位外部数据总线。Pentium 芯片的内部 ALU 和通用寄存器仍是 32 位,所以还是 32 位微处理器,但它同内存储器进行数据交换采用了 64 位的外部数据总线,两者之间的数据传输速率可达 528MB/s。

2.2.3 32 位微处理器的编程结构

对汇编语言程序员而言,掌握所用微处理器的寄存器结构是至关重要的,因为在这些寄存器中,有的是程序设计期间必须使用的,被称为程序员可见的寄存器;有的虽然在应用程序设计期间,不能直接寻址,被称为程序不可见的寄存器,但这些寄存器在系统运行程序期间可能被间接用于控制和操作保护模式下的存储器系统。

80x86 微处理器的寄存器组主要包括以下几个部分:基本结构寄存器(Base Architecture Registers)、系统级寄存器(System Level Registers)、调试和测试寄存器(Debug and Test Registers)和浮点寄存器(Floating Point Registers)。

1. 基本结构寄存器

图 2-2 表示了 80x86 微处理器的基本体系结构寄存器,这些寄存器在用汇编语言编写程序时都可以访问。Pentium 与 80386 相比,除了标志寄存器外,其余寄存器的命名和使用方法都没有改变。

(1) 通用寄存器

8 个 32 位的通用寄存器如图 2-2(a)所示,这些寄存器都可以存放数据或地址,并能进行 32 位、16 位、8 位和 1 位的运算。

能进行 32 位运算的寄存器分别称为 EAX、EBX、ECX、EDX、ESI、EDI、EBP 和 ESP。

这 8 个寄存器的低 16 位可独立使用,它们分别以 AX、BX、CX、DX、SI、DI、BP 和 SP 为名被访问。其中 AX、BX、CX、DX 的低位字节或高位字节也可作为独立的 8 位寄存器使用,低位字节的寄存器分别称为 AL、BL、CL、DL,高位字节的寄存器分别称为 AH、BH、CH、DH。

(2) 指令指针

指令指针如图 2-2(b)所示,它是 32 位的寄存器,称为 EIP。EIP 中存放相对于代码段寄存器(CS)的基址的偏移量。EIP 的低 16 位可作独立使用的寄存器,称为 IP,它在实

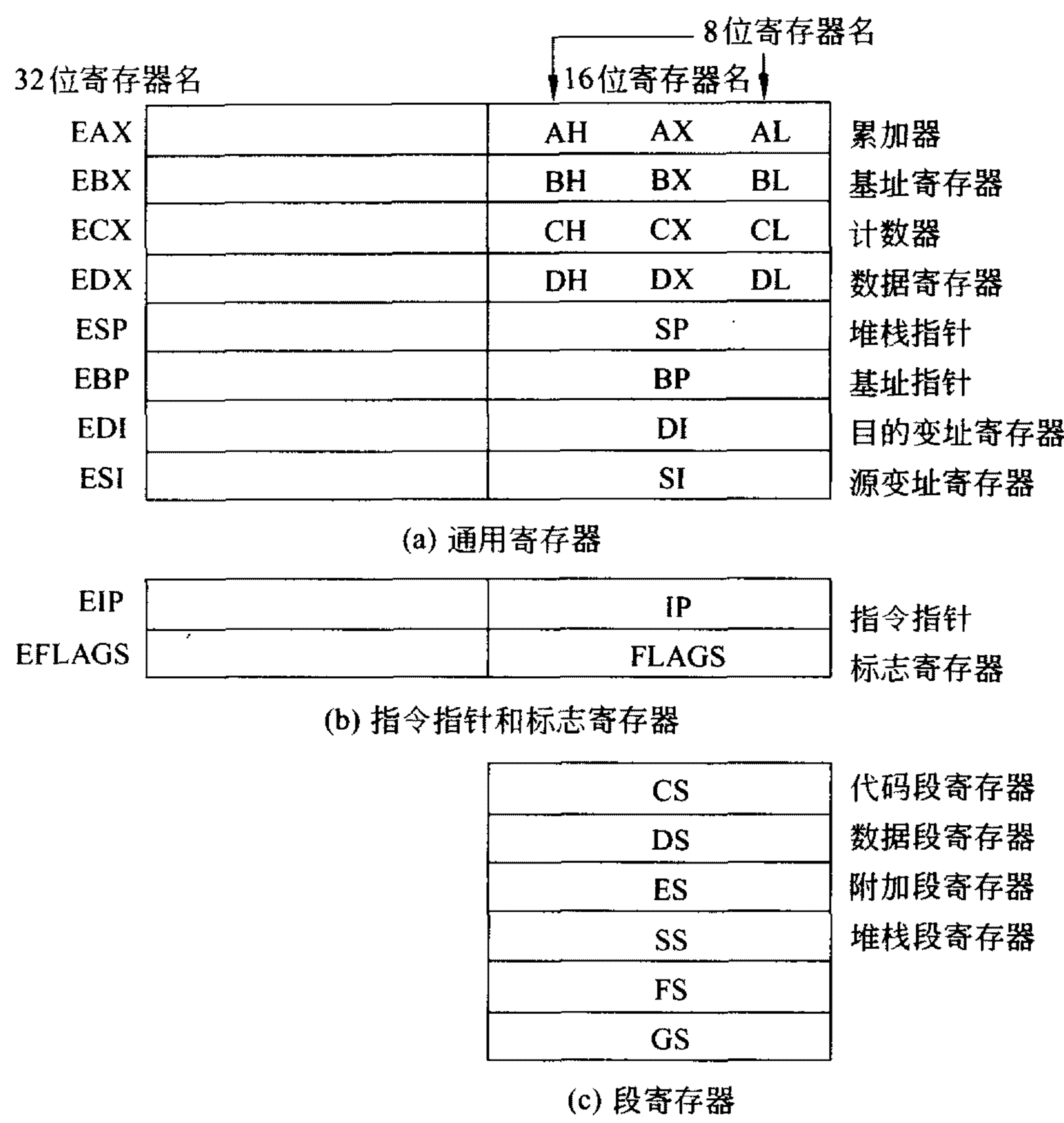


图 2-2 基本体系结构寄存器

地址模式时,与 CS 组合后,形成 20 位的物理地址。

(3) 标志寄存器

标志寄存器见图 2-2(b),它是 32 位的寄存器,称为 EFLAGS。EFLAGS 中的位可分为标志位和控制位二类,标志位指明程序运行时的微处理器的实时状态;控制位由程序设计者设置,以控制 CPU 进行某种操作。EFLAGS 的低位也可作为一个独立的标志寄存器使用,它被称为 FLAGS。它在实地址模式时是最有用的。

EFLAGS 中各位的定义在有关章节介绍。

(4) 段寄存器

设计程序时,一般把指令代码和数据分开保存于不同的存储器空间。80x86 微处理器内部有 6 个 16 位的段寄存器用于指示代码和数据所用的地址空间,它们是代码段寄存器 CS、堆栈段寄存器 SS,DS、ES、FS 和 GS 都称为数据段寄存器,见图 2-2(c)。除 CS 是用于指示指令代码的地址空间之外,其他段寄存器都用于指示数据的地址空间。当微处理器工作在实地址方式时,这些段寄存器提供的内容就是 16 位的段基址。

2. 系统级寄存器

系统级寄存器包含了控制寄存器、系统地址寄存器和调试测试寄存器三类。Pentium 与 80486 相比,其地址和调试寄存器的命名方法以及功能都相同,其他寄存器则

是有区别的。这些寄存器控制着 80x86 微处理器的片内 Cache、运算部分的浮点部件以及存储管理部分。它们只在系统程序中才能使用。

(1) 控制寄存器

Pentium 微处理器的控制寄存器如图 2-3 所示。CR0、CR1、CR2、CR3 和 CR4 是 32 位的控制寄存器,其中 CR1 是 Intel 公司为以后开发的微处理器而保留的控制寄存器。

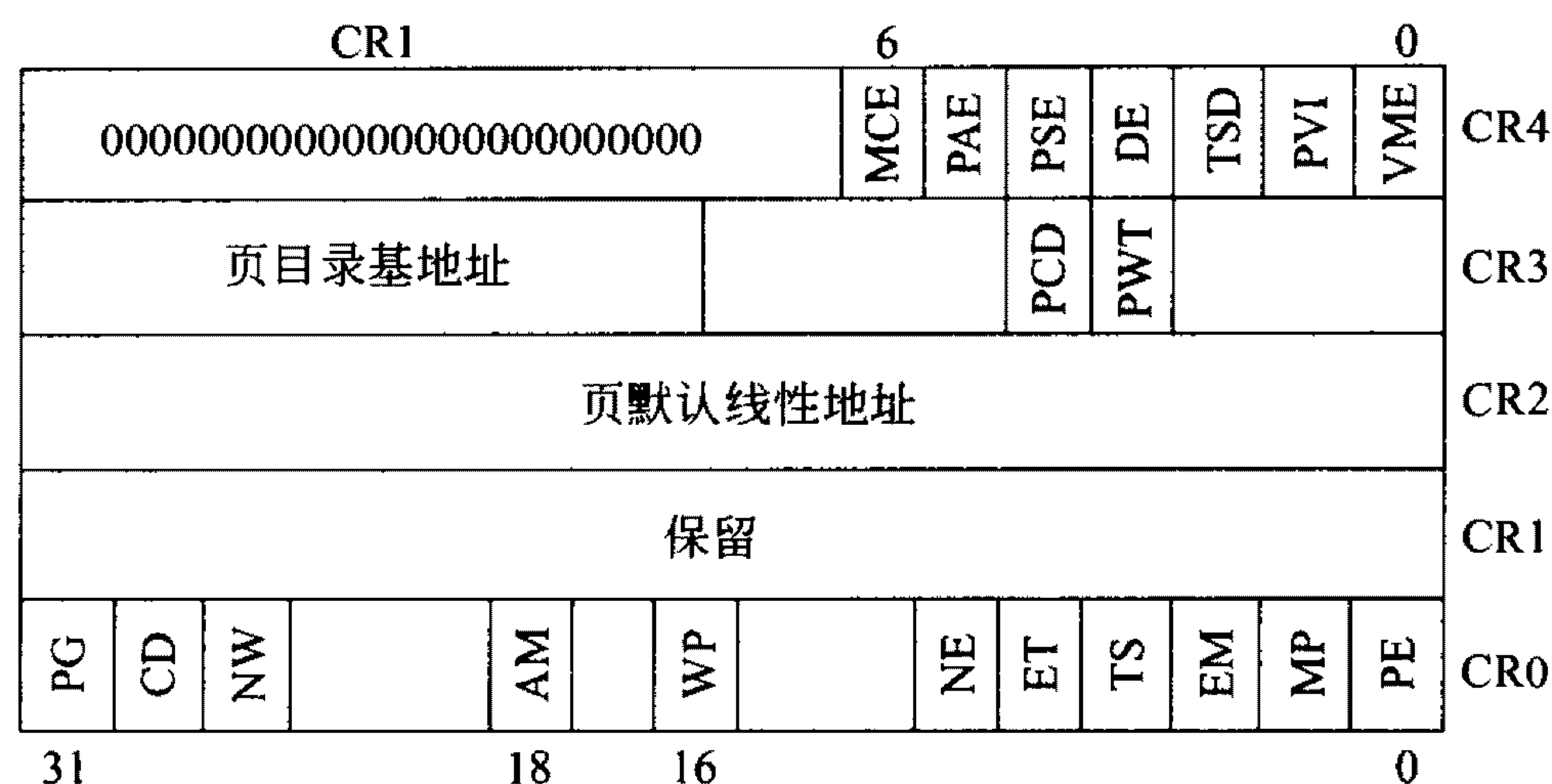


图 2-3 Pentium 控制寄存器结构

• CR0 介绍

Intel 公司对 CR0 中的 11 位进行了定义,剩余的 21 位是 Intel 公司自己保留的。

PE(第 0 位)是保护虚拟地址方式的允许位。当 PE=0 时,80x86 微处理器工作于保护虚拟地址方式;反之工作于实地址方式。

PG(第 31 位)是分页允许位。当 PG=1 时,允许分页部件工作;反之禁止分页部件工作。允许分页部件工作的前提是微处理器必须处于保护虚拟地址方式下,即 PE=1。

CD(第 30 位)是片内 Cache 的无效位。当 CD=1 时,片内 Cache 不命中,则不把所需信息读入片内 Cache;当 CD=0 时,片内 Cache 不命中,则把所需信息读入片内 Cache。

NW(第 29 位)是片内 Cache 非写直达位。当 NW=0 时,在数据写入片内 Cache 的同时也写入主存;当 NW=1 时,数据仅写入片内 Cache。

WP(第 16 位)是页写保护位。当 WP=1 时,禁止任何特权级的程序对只读页面进行写入操作。

AM(第 18 位)是对界检查控制位。

NE(第 5 位)是数值异常位;TS(第 3 位)是任务切换位;EM(第 2 位)是仿真协处理器位;MP(第 1 位)是监视协处理器位,这 4 位都与浮点部件的控制有关,本书不介绍浮点运算,这 4 位的具体定义请参阅有关资料。

• CR2 介绍

CR2 保留了所检测到的上一个页面故障的 32 位线性地址。

• CR3 介绍

CR3 的 20 位(第 12~31 位)保存着一级页表(页目录)的物理基地址。

CR3 还可对外部 Cache(二级 Cache)进行控制,这是由 PWT(第 3 位)和 PCD(第 4 位)进行的。

• CR4 介绍

CR4 用到第 6 位,其余均为 0。

VME(第 0 位)是虚拟 8086 模式中断允许位。在虚拟 8086 模式下,此位为 1,则允许中断,为 0 则禁止。

PVI(第 1 位)是保护模式虚拟中断允许位。在保护模式下,此位为 1,则允许中断,为 0 则禁止。

TSD(第 2 位)是读时间计数器指令的特权设置位。只有此位为 1,才能使读时间计数器指令 RDTSC 作为特权指令可在任何时候执行,否则仅允许在系统级执行。

DE(第 3 位)是断点有效允许位。该位为 1 则支持断点设置,为 0 则禁止。

PSE(第 4 位)是页面扩展允许位。该位为 1 则页面尺寸为 4MB,否则为 4KB。

PAE(第 5 位)是物理地址扩充允许位。该位为 1 则允许按 36 位物理地址运行分页机制,否则按 32 位运行分页机制。

MCE(第 6 位)是机器检查允许位。此位为 1 则使机器检查异常功能有效。

(2) 系统地址寄存器

系统地址寄存器又称为保护方式寄存器。顾名思义,它们仅能在保护方式下使用。系统地址寄存器有 4 个,如图 2-4 所示。它们分别称为全局描述符表寄存器(GDTR)、中断描述符表寄存器(IDTR)、局部描述符表寄存器(LDTR)和任务状态寄存器(TR)。

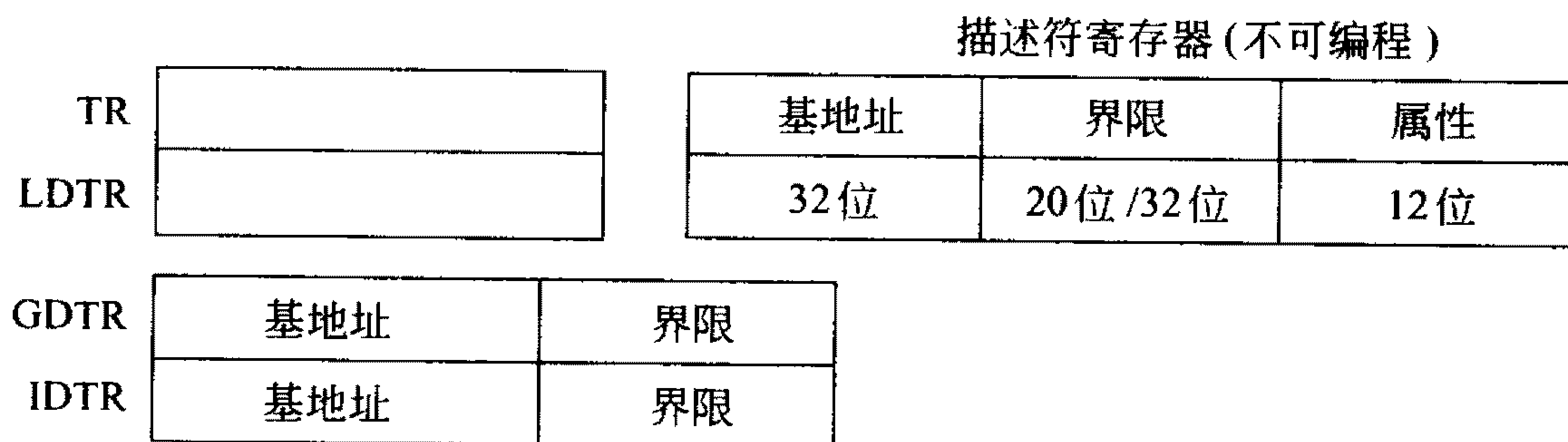


图 2-4 系统地址寄存器

• GDTR

GDTR 是 48 位寄存器,其中高 32 位是全局描述符表(GDT)的线性基地址,低 16 位是界限——全局描述符表的尺寸。

例:若(GDTR)=0800,0000,1000H,则全局描述符表线性基地址为 0800,0000H,其最后一个地址为 08000FFFH。

• IDTR

IDTR 也是 48 位的寄存器,其中高 32 位是中断描述符表(IDT)的线性基地址,低 16 位是 IDT 的界限(尺寸)。

• LDTR

该寄存器用于存放局部描述符表(LDT)的线性基地址、界限、描述符的属性和 16 位的选择符。除 16 位的选择符可由程序访问外,其他部分都是不可见的,它们都由硬件自动装入内容,因此在指令系统中所用到的 LDTR 只是指 16 位的选择符。

• TR

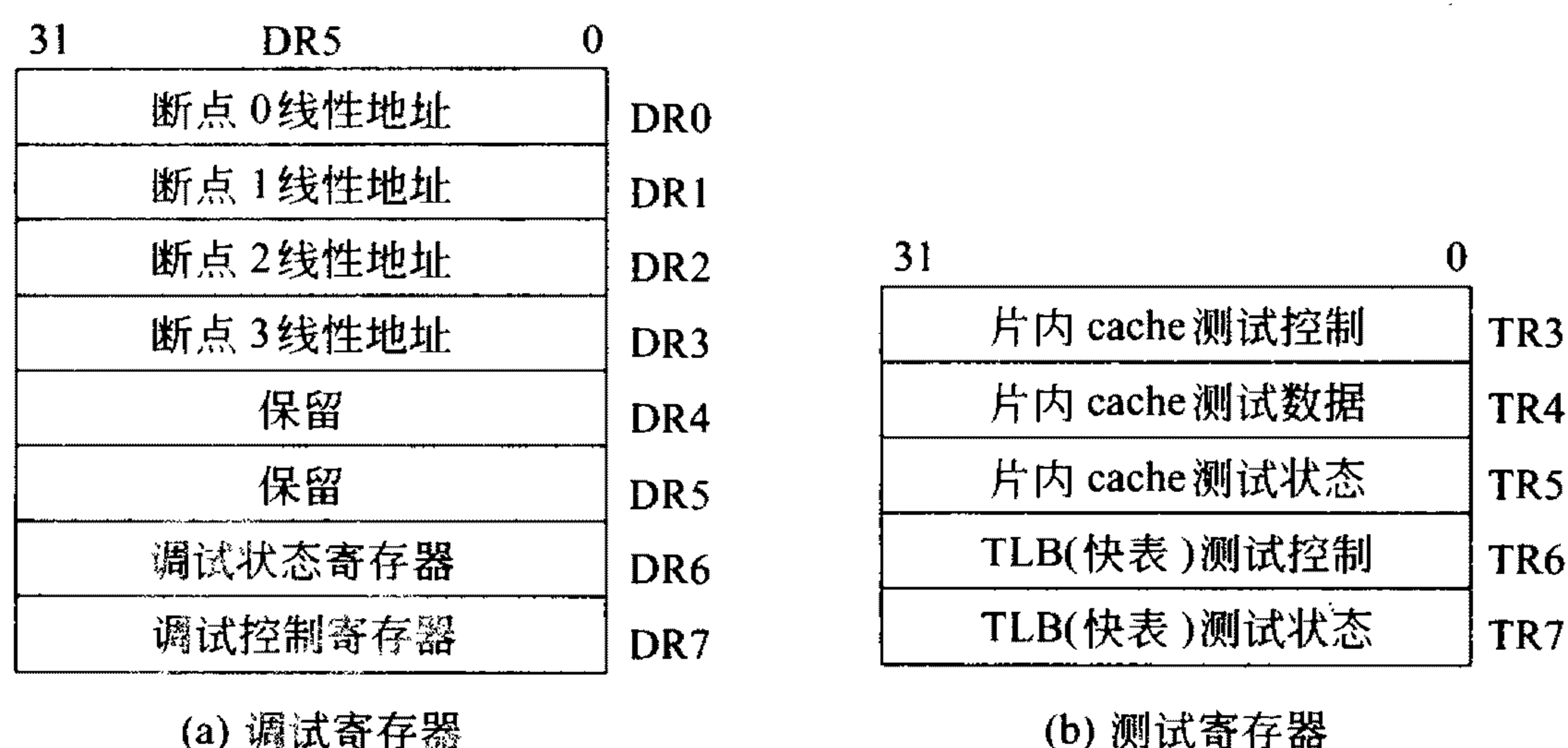
该寄存器用于存放当前正在执行的任务的线性基地址、界限、描述符的属性和 16 位

的选择符。与 LDTR 类似,只有 16 位的选择符可由程序访问,因此指令系统中所用到的 TR 只是指 16 位的选择符。

选择符和各种描述符及描述符表在有关章节介绍。

(3) 调试与测试寄存器

80x86 微处理器有 8 个调试寄存器,如图 2-5(a)所示。它们对程序的调试提供了硬件上的支持。DR0~DR3 用于设置数据存取断点和代码执行断点;DR7 是调试控制寄存器,用于选择调试功能和设置断点;DR6 是调试状态寄存器,它主要用于指明断点的当前状态。DR4 和 DR5 是 Intel 公司为 80486 之后所要开发的微处理器而保留的。



(a) 调试寄存器

(b) 测试寄存器

图 2-5 调试与测试寄存器

80486 微处理器还包含 5 个测试寄存器,它们用于测试自身的片内 Cache 和转换后备缓冲区 TLB,见图 2-5(b)。而 Pentium 则取消了测试寄存器,用一组模式专用寄存器来实现更多的功能。模式专用寄存器的含义如表 2-1 所示。

表 2-1 Pentium 模式专用寄存器的含义

ECX	寄存器名	其中内容
00H	机器检查地址	引起异常周期的存储器单元的地址
01H	机器检查类型	引起异常周期的总线周期类型
02H	测试寄存器 1	测试奇偶校验错误
03H	保留	
04H	测试寄存器 2	测试指令 Cache 的结束位,含 4 位,每位对应 2 个双字中的 1 个字节,为 1 则表示对应字节为指令结束字节
05H	测试寄存器 3	Cache 的数据测试,保存读写的双字数据
06H	测试寄存器 4	提供 Cache 的有效域、有效位和标签域
07H	测试寄存器 5	提供 Cache 的组选择域等参数
08H	测试寄存器 6	TLB 线性地址测试,含 31~12 位
09H	测试寄存器 7	TLB 物理地址测试,含 31~12 位
0AH	测试寄存器 8	TLB 物理地址测试,含 35~32 位
0BH	测试寄存器 9	BTB 标签测试,含标签地址和历史信息
0CH	测试寄存器 10	BTB 目标测试,含线性地址
0DH	测试寄存器 11	BTB 命令测试,含读写命令
0EH	测试寄存器 12	允许跟踪和分支预测

续表

ECX	寄存器名	其中内容
0FH	保留	
10H	时间标志计数器	性能监测
11H	控制和选择	性能监测
12H	计数器 0	性能监测
13H	计数器 1	性能监测
14H	保留	

这些寄存器中,寄存器 00H 和 01H 是 64 位的,读写的内容在 EDX~EAX 中,其余寄存器均为 32 位,读写的内容在 EAX 中。

3. 浮点寄存器组

浮点寄存器组是 Pentium 以上的 32 位微处理器内部所有的,因为它内部包含了浮点运算部件 FPU。与之匹配的有 8 个数据寄存器、1 个标记字寄存器、1 个状态寄存器、1 个控制寄存器、1 个指令指针寄存器和 1 个数据指针寄存器。

(1) 数据寄存器

8 个数据寄存器 R0~R7,每个为 80 位,相当于 20 个 32 位寄存器。每个 80 位寄存器中,1 位为符号位,15 位为阶码,64 位为尾数,以此对应浮点运算时扩展精度的数据类型。

(2) 标记字寄存器

这是 1 个 16 位的寄存器,用标记来指示数据寄存器的状态,如图 2-6 所示,每个数据寄存器对应标记字寄存器中的两位,数据寄存器 R0 对应标记字寄存器中的 1 位、0 位,依此类推,R7 对应标记字寄存器中的 15 位、14 位。通过标记字来表示对应数据寄存器是否为空,这种功能可使 FPU 更加简洁地对数据寄存器作检测。

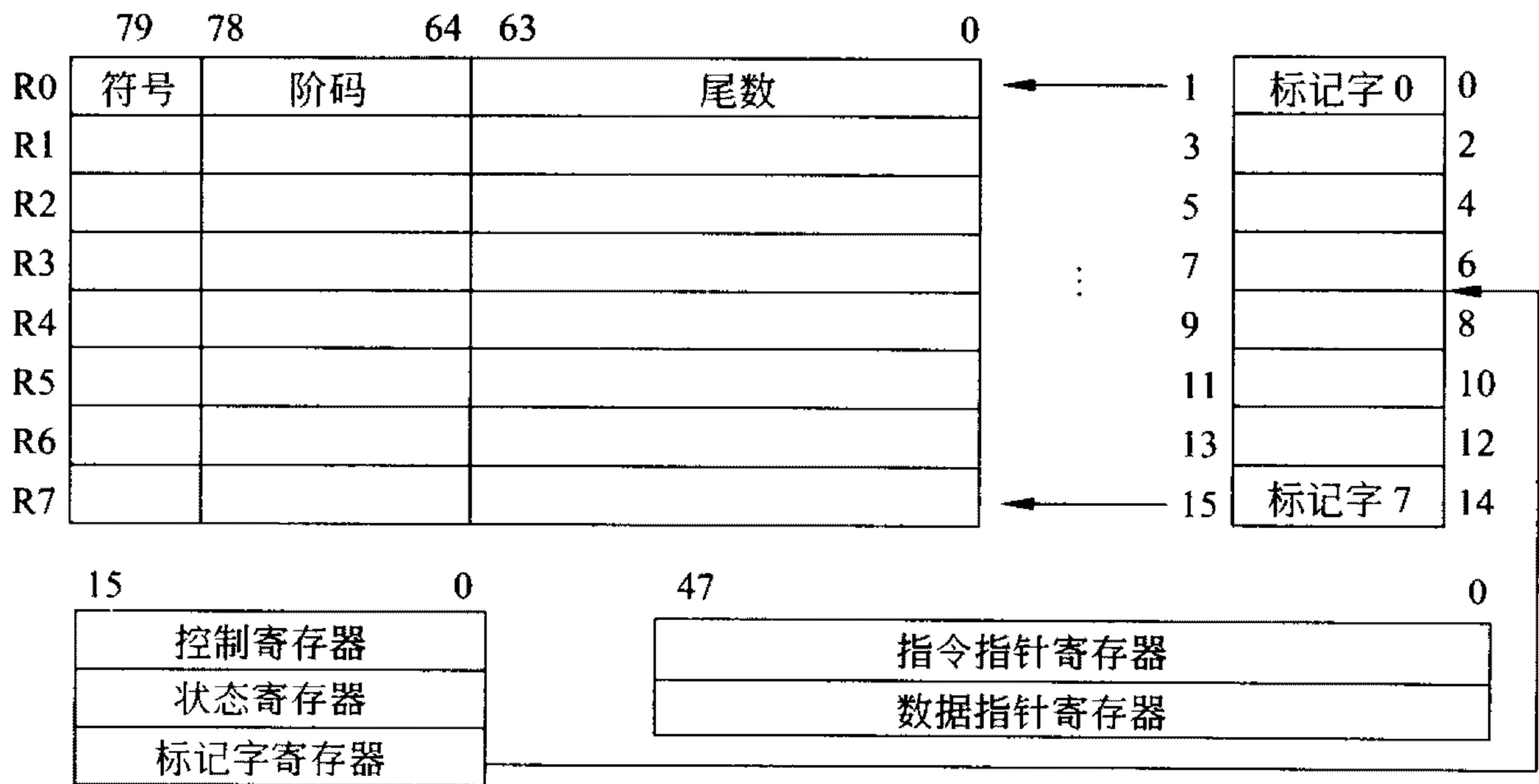


图 2-6 Pentium 的浮点寄存器组

(3) 状态寄存器

16 位的状态寄存器用来指示 FPU 的当前状态,如图 2-7 所示。

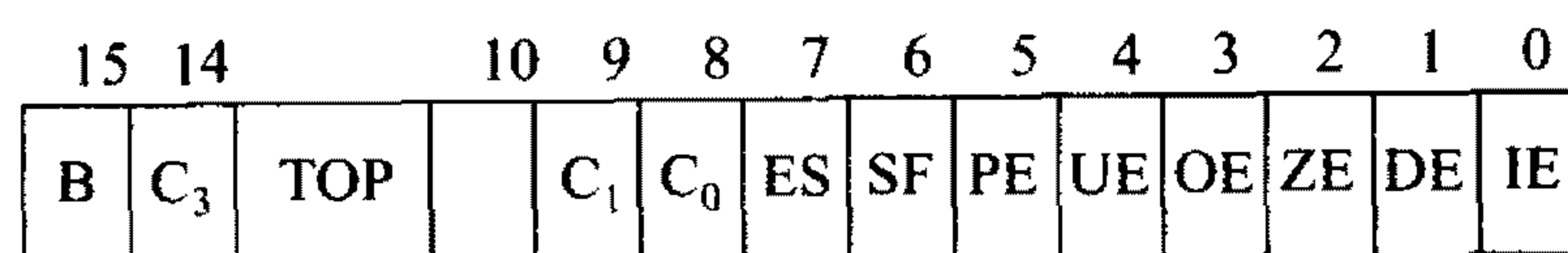


图 2-7 Pentium 的 FPU 状态寄存器

IE 表示无效操作,这是非法操作引起的故障。

DE 为 1 表示操作数不符合规范引起故障。

ZE 为 1 表示除数为 0 引起故障。

UE 和 OE 分别表示浮点运算出现上溢和下溢。

PE 为 1 表示运算结果不符合精度规格。

SF 是堆栈故障标志,当 $IE=1$ 且 $SF=1$ 时,若 $C_1=1$,则表示堆栈上溢引起无效操作;若 $C_1=0$,则表示堆栈下溢引起无效操作。

ES 为错误标志,上面任何一个故障都会同时使 $ES=1$,且使 \overline{FERR} 信号为低电平。

$C_0 \sim C_3$ 被称为条件码,除了 C_1 和 SF 一起表示堆栈状态外,这几个代码一方面可以用 SAHF 指令进行设置,另一方面,可用 FSTSW AX 指令读取,然后,以此为条件实现某种选择,条件码之名正是由此而来。

TOP 是栈顶指针。

B 位用来指示浮点运算器的当前状态,为 1 表示处于忙状态。

(4) 控制字寄存器

如图 2-8 所示,控制寄存器的低 6 位分别用来对 6 种异常进行屏蔽,这些屏蔽位和状态寄存器的标志位一一对应。比如,IM 对应 IE,DM 对应 DE……

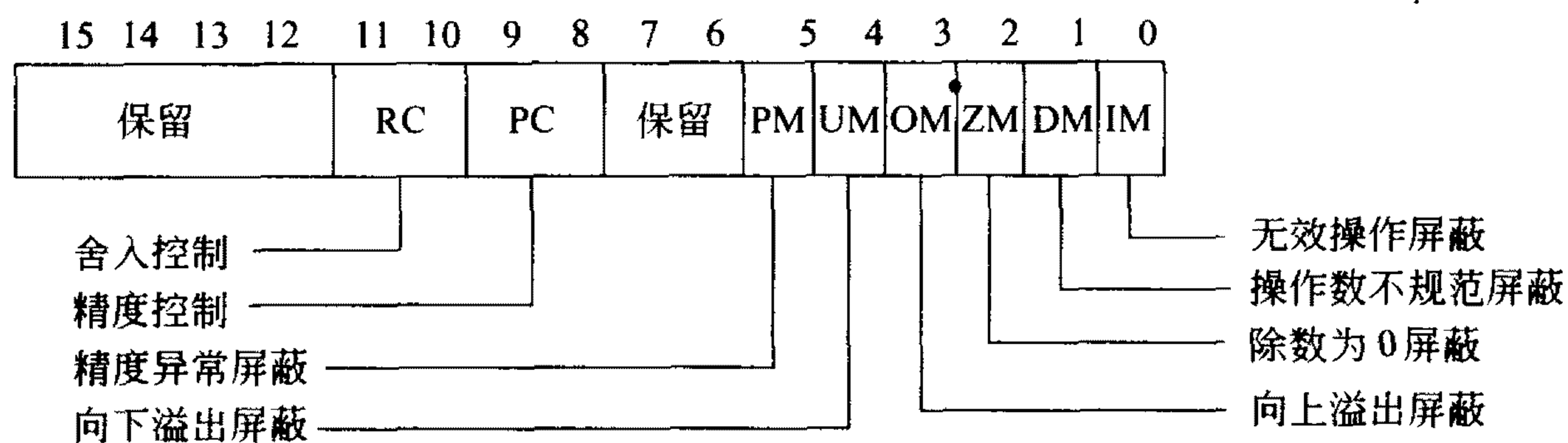


图 2-8 Pentium 的 FPU 控制字寄存器

PC 占 2 位,用作精度控制,可选 24 位单精度(00),53 位双精度(10)和 64 位扩展双精度(11)或保留(01)。

RC 也占 2 位,用作舍入控制,可设置为靠近偶数舍入(00)、向下舍入(01)、向上舍入(10)和截断舍入(11)。

这些舍入方式的含义如下所述。比如,浮点运算结果为 x ,与 x 最靠近的两个数为 m 和 n ,并且, $n < x < m$,靠近偶数舍入的含义是指将 x 舍入为 m 和 n 中末位为 0 的那个偶数,如两个均为偶数,则取差值偏小者;向下舍入的含义是指将 x 舍入为 n ,这是指往 1 方

向舍入;向上舍入的含义是将 x 舍入为 m , 指往 $+\infty$ 方向舍入;截断舍入的含义是从 m 和 n 中选择绝对值小的那个数作为舍入值, 这是指往 0 方向舍入。

(5) 指令指针和数据指针寄存器

这两个寄存器用来提供发生故障的指令的地址及其数据操作对应的存储器的地址。

2.3 32 位微处理器的外部引脚

理解微处理器的引脚功能是学习微型计算机系统中存储器接口和 I/O 接口的基础, Pentium 由于增加了许多功能, 使得信号数量大大增加, 芯片有 168 个引脚, 本节主要介绍 Pentium 微处理器的引脚功能。

1. 地址线及控制信号

$A_{31} \sim A_3$	地址线。
AP	地址的偶校验码位。
\overline{ADS}	地址状态输出信号。
$\overline{A_{20}M}$	A_{20} 以上的地址线屏蔽信号。
\overline{APCHK}	地址校验出错信号。

由于 Pentium 有片内 Cache, 所以地址线是双向的, 既能对外选择主存和 I/O 设备, 也能对内选择片内 Cache 的单元。Pentium 的 32 位地址线可寻址 4GB 内存和 64KB 的 I/O 空间, 32 位地址信号中, 低 3 位地址 $A_2 \sim A_0$ 组合成字节允许信号 $\overline{BE}_7 \sim \overline{BE}_0$, 所以, $A_2 \sim A_0$ 不对外。

当 $A_{31} \sim A_3$ 有输出时, AP 上输出偶校验码, 供存储器对地址进行校验。在读取 Cache 时, Pentium 会对地址进行偶校验, 如校验有错, 则地址校验信号 \overline{APCHK} 输出低电平。 \overline{ADS} 为地址状态信号, 它表示 CPU 已启动 1 个总线周期。

$\overline{A_{20}M}$ 信号是所有与 ISA 总线兼容的计算机系统中必须有的信号, 当此信号为 0 时, 将屏蔽第 20 位以上的地址, 以便在访问 Cache 和主存时可仿真 1MB 存储空间。

2. 数据线及控制信号

$D_{63} \sim D_0$	数据线。
$\overline{BE}_7 \sim \overline{BE}_0$	字节允许信号。
$DP_7 \sim DP_0$	奇偶校验信号。
\overline{PCHK}	读校验出错。
\overline{PEN}	奇偶校验允许信号, 若输入为低电平, 则在读校验出错时处理器会自动作异常处理。

Pentium 对外用 64 位数据线, 所以数据总线为 $D_{63} \sim D_0$, 并增加了奇偶校验, 在对存储器进行读写时, 每个字节产生 1 个校验位, 通过 $DP_7 \sim DP_0$ 输出, 而读操作时, 则按字节

进行校验。 $\overline{\text{PCHK}}$ 信号在读校验出错时为 0, 这样便送外部电路告示校验出错。

$\overline{\text{BE}}_7 \sim \overline{\text{BE}}_0$ 为字节允许信号, 对应 8 字节即 64 位数据。

3. 总线周期控制信号

- | | |
|---------------------------------|--|
| $\text{D}/\overline{\text{C}}$ | 数据/控制信号。高电平表示当前总线周期传输的是数据, 低电平表示当前总线周期传输的是指令。 |
| $\text{M}/\overline{\text{IO}}$ | 存储器和 I/O 访问信号。高电平时访问存储器, 低电平时则访问 I/O 端口。 |
| $\text{W}/\overline{\text{R}}$ | 读写信号。高电平时表示当前总线周期进行写操作, 低电平时则为读操作。 |
| $\overline{\text{LOCK}}$ | 总线封锁信号。低电平有效, 此时将总线锁定, $\overline{\text{LOCK}}$ 信号由 LOCK 指令的前缀设置, 总线被锁定时使得其他总线主设备不能获得总线控制权, 从而确保 CPU 完成当前操作。 |
| $\overline{\text{BRDY}}$ | 突发就绪信号。表示结束一个突发总线传输周期, 此时外设处于准备好的状态。 |
| $\overline{\text{NA}}$ | 下一个地址有效信号。低电平有效, 从此端输入低电平时, CPU 会在当前总线周期完成之前就将下一个地址送到总线上, 从而开始下一个总线周期, 构成所谓总线流水线工作方式, Pentium 允许 2 个总线周期构成总线流水线。 |
| SCYC | 分割周期信号。表示当前地址指针未对准字、双字或四字的起始字节, 因此, 要采用 2 个总线周期完成数据传输, 即对周期进行分割。 |

$\text{M}/\overline{\text{IO}}$ 、 $\text{D}/\overline{\text{C}}$ 和 $\text{W}/\overline{\text{R}}$ 信号和 80386 的对应信号相同。 $\overline{\text{BRDY}}$ 和 $\overline{\text{RDY}}$ 信号类似, $\overline{\text{RDY}}$ 信号有效, 表示结束 1 个普通传输周期, $\overline{\text{BRDY}}$ 有效, 表示结束 1 个突发传输周期。在 $\overline{\text{RDY}}$ 和 $\overline{\text{BRDY}}$ 均有效时, CPU 会忽略 $\overline{\text{BRDY}}$ 。

4. Cache 控制信号

- | | |
|---------------------------|--|
| $\overline{\text{CACHE}}$ | Cache 控制信号。在读操作时, 如此信号输出低电平, 表示主存中读取的数据正在送入 Cache; 写操作时, 如此信号为低电平, 表示 Cache 中修改过的数据正写回到主存。 |
| $\overline{\text{EADS}}$ | 外部地址有效信号。此信号为低电平时外部地址有效, 此时可访问片内 Cache。 |
| $\overline{\text{KEN}}$ | Cache 允许信号。确定当前总线周期传输的数据是否送到 Cache。 |
| $\overline{\text{FLUSH}}$ | Cache 擦除信号。此信号有效时, CPU 强制对片内 Cache 中修改过的数据回写到主存, 然后擦除 Cache。 |
| AHOLD | 地址保持/请求信号。高电平有效, 用以强制 CPU 浮空地址信号, 为从 $\text{A}_{31} \sim \text{A}_4$ 输入地址访问 Cache 作准备。 |

- PCD Cache 禁止信号。高电平时,禁止对片外 Cache 的访问。
- PWT 片外 Cache 的控制信号。高电平时使 Cache 为通写方式,低电平时为回写方式。
- WB/ $\overline{\text{WT}}$ 片内 Cache 回写/通写选择信号。此信号为 1,则为回写方式,为 0 则为通写方式。
- $\overline{\text{HIT}}$ 和 $\overline{\text{HITM}}$ Cache 命中信号和命中 Cache 的状态信号。 $\overline{\text{HIT}}$ 低电平时,表示 Cache 被命中。 $\overline{\text{HITM}}$ 低电平时,表示命中的 Cache 被修改过。
- INV 无效请求信号。此信号为高电平时,使 Cache 区域不可再使用而成为无效。

如外部存储器子系统将 $\overline{\text{KEN}}$ 信号设置为低电平,就会在存储器读周期中将数据复制到 Cache。

PCD 和 PWT 是用来控制片外 Cache 的。PCD 信号用来向外接 Cache 告示,当前访问的页面已在片内 Cache 中,所以,不必启用外接 Cache。PWT 信号有效时,对外接 Cache 按通写方式操作,否则按回写方式操作。

AHOLD 和 $\overline{\text{EADS}}$ 信号用来保证 Cache 数据的一致性。这种情况发生在 DMA 传输中,主存和外设直接交换数据,当主存中某个数据被修改时,如这两个信号有效,Pentium 会马上检查此处原来的数据是否在 Cache 中,如果是,则应使 Cache 中的数据无效,以保证数据的正确性和一致性。为此,主存系统在写操作后,通过外部电路将 AHOLD 置 1, Pentium 收到此信号以后,使地址处于高阻状态即无效状态,然后,外部电路把被修改单元的地址送到地址总线,并将 $\overline{\text{EADS}}$ 置 0,使外部地址有效,此时,Cache 系统如果检测到 Cache 有此地址的数据,则会作无效处理,从而保证 Cache 内和主存内的数据一致。

$\overline{\text{HIT}}$ 、 $\overline{\text{HITM}}$ 和 INV 用于一种特殊的称为询问周期的操作,在这种总线周期中,通过一个专用端口查询数据 Cache 和指令 Cache,以确定当前地址是否命中 Cache,如果命中,则 $\overline{\text{HIT}}$ 为低电平,如果不但命中 Cache,而且此数据已修改过,则 $\overline{\text{HITM}}$ 也为低电平。而 INV 端输入高电平时,使 Cache 不能访问。

5. 系统控制信号

- INTR 可屏蔽中断请求信号。
- NMI 非屏蔽中断请求信号。
- RESET 系统复位信号。
- INIT 初始化信号。
- CLK 系统时钟信号。

INIT 信号和 RESET 信号类似,都用于对 CPU 处理器作初始化,但两者有区别。RESET 有效时,会使处理器在 2 个时钟周期内终止程序,即进行复位,而 INIT 有效时,处理器先将此信号锁存,直到当前指令结束时才执行复位操作。另外,用 INIT 信号复位时,只对基本寄存器进行初始化,而 Cache 和浮点寄存器中的内容不变。但不管是用 RESET 信号还是用 INIT 信号,系统复位以后,程序均从 FFFFFFFF0H 处重新开始运行。复位后,微处理器内部寄存器值见表 2-2。

表 2-2 复位后寄存器的值

寄存器	初始值	寄存器	初始值
EAX	不定	ES	0000H
EBX	不定	CS	F000H
ECX	不定	SS	0000H
EDX	0400H+版本 ID	DS	0000H
EBP	不定	FS	0000H
ESP	不定	GS	0000H
EDI	不定	IDTR	基址=0,界限=3FFH
ESI	不定	CR ₀	6000-0000H
EFLAGS	0000-0002H	DR7	0000-0000H
EIP	0FFF0H	浮点寄存器	不变

6. 总线仲裁信号

- HOLD** 总线请求信号。这是其他总线主设备请求 CPU 让出总线控制权的信号。
- HLDA** 总线请求响应信号。这是对 HOLD 的回答信号,表示 CPU 已让出总线控制权。
- BREQ** 总线周期请求信号。此信号有效时,向其他总线主设备告示,CPU 当前已提出一个总线请求,并正在占用总线。
- $\overline{\text{BOFF}}$** 强制让出总线信号。此信号强制 CPU 让出总线控制权,CPU 接到此信号时,立即放弃总线控制权,直至此信号无效时,CPU 再启动被打断的总线周期。

$\overline{\text{BOFF}}$ 和 HOLD 有类似之处,但有两点不同:一是 $\overline{\text{BOFF}}$ 会使当前时钟周期一结束即让出总线控制权,此时总线周期并没结束,而 HOLD 则在当前总线周期结束时才让出总线控制权,所以可能还会持续 1 个或几个时钟周期,动作较慢;二是 $\overline{\text{BOFF}}$ 没有对应的响应信号,而 HOLD 有响应信号 HLDA。外部总线主设备可用 $\overline{\text{BOFF}}$ 信号快速获得总线控制权。

7. 检测与处理信号

- $\overline{\text{BUSCHK}}$** 转入异常处理的信号。
- $\overline{\text{FERR}}$** 浮点运算出错的信号。
- $\overline{\text{IGNNE}}$** 忽略浮点运算错误的信号。低电平有效,此时 CPU 会忽略浮点运算错误。
- $\overline{\text{FRCMC}}$** 输入此信号会使 CPU 进行冗余校验。
- $\overline{\text{IERR}}$** 冗余校验出错信号。与 $\overline{\text{FRCMC}}$ 配合使用,此信号有效表示冗余校验出错。

$\overline{\text{BUSCHK}}$ 信号由外部电路输入,外部电路在检测到当前总线周期未正常结束时,将

此信号置于低电平,此后,CPU 会采样此信号,如为低电平,则使当前错误总线周期结束并转入异常处理。

CPU 在 RESET 信号由高到低时,对 $\overline{\text{FRCMC}}$ 已采样,如采样到低电平,则 CPU 进入冗余校验状态,如校验出错,则 $\overline{\text{IERR}}$ 输出低电平。

8. 系统管理模式信号

$\overline{\text{SMI}}$ 系统管理模式中断请求信号。这是对进入系统管理模式的中断请求。

$\overline{\text{SMIACT}}$ 系统管理模式信号。这是对 $\overline{\text{SMI}}$ 信号的响应信号,当 $\overline{\text{SMI}}$ 中断请求有效时,CPU 输出 $\overline{\text{SMIACT}}$ 表示中断请求成功,当前已处于系统管理模式。

$\overline{\text{SMI}}$ 用来进入系统管理模式,要退出系统管理模式时,可用 RSM 指令。

9. 测试信号

TCK 从此端输入测试时钟信号。

TDI 用来输入串行测试数据。

TDO 此端获得输出的测试数据结果。

TMS 用来选择测试方式。

TRST 测试复位,退出测试状态。

10. 跟踪和检查信号

$\text{BP}_3 \sim \text{BP}_0$ 以及 $\text{PM}_1 \sim \text{PM}_0$ 、 $\text{BP}_3 \sim \text{BP}_0$ 是与调试寄存器 $\text{DR}_3 \sim \text{DR}_0$ 中的断点相匹配的外部输出信号, $\text{PM}_1 \sim \text{PM}_0$ 是性能监测信号。

$\text{BT}_3 \sim \text{BT}_0$ 分支地址输出信号, $\text{BT}_2 \sim \text{BT}_0$ 上输出分支地址的最低 3 位。

IU 高电平有效,表示此时 U 流水线完成指令的执行过程。

IV 高电平有效,表示此时 V 流水线完成指令的执行过程。

IBT 指令发生分支。

$\text{R}/\overline{\text{S}}$ 探针信号输入端,此信号从高到低的跳变会使处理器停止执行指令进入空闲状态。

PRDY 这是对 $\text{R}/\overline{\text{S}}$ 的响应信号,输出高电平时表示 CPU 当前停止执行指令,从而可以进入测试。

IU、IV、IBT 都是输出信号,可通过对其电平的检测来跟踪指令的执行。 $\text{PM}_1 \sim \text{PM}_0$ 和 $\text{BP}_1 \sim \text{BP}_0$ 是复用的,由调试寄存器 DR_7 中的 GE 和 LE 两位确定,如两者为 1,则为 $\text{BP}_1 \sim \text{BP}_0$,否则为 $\text{PM}_1 \sim \text{PM}_0$ 。

2.4 32 位微处理器的工作模式

80486 等 32 位微处理器有三种工作模式(工作方式),一种是实地址模式(Real Address Mode),一种是保护虚拟地址模式(Protected Virtual Address Mode),也叫保护模式,还有一种叫虚拟 8086 模式。我们先介绍 80x86 的地址空间,然后就这三种模式进行阐述。

2.4.1 80x86 的地址空间

1. 存储地址空间

80x86 微处理器为存储系统、特别是为虚拟存储器的实现提供了有力的支持。80x86 微处理器有 3 个明确的存储地址空间,它们是虚拟空间、线性空间和物理空间。

虚拟空间又称逻辑空间,是应用程序员编写程序的空间,其相应的地址称为虚拟地址或逻辑地址。这个空间非常大,32 位微处理器的逻辑空间可达 2^{46} B(64TB)。

物理空间也称主存空间,是计算机中主存储器的实际空间,相应的地址称为物理地址或主存地址。32 位微处理器能访问的主存空间为 2^{32} B(4GB)。

32 位微处理器通过分段部件把虚拟空间变换为 32 位的线性空间,如果分页部件未被选用,线性地址就是物理地址,如图 2-9 所示。

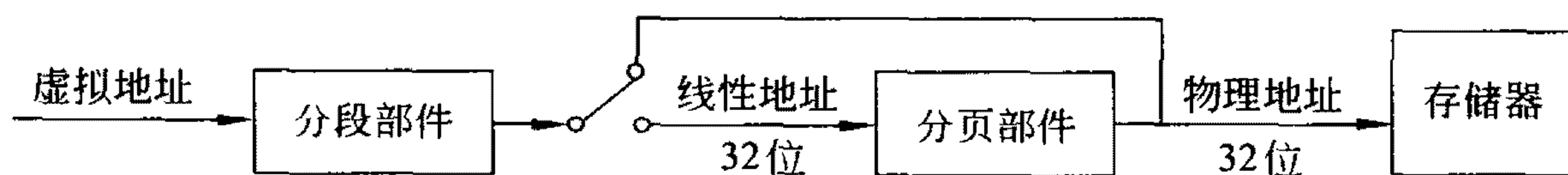


图 2-9 地址变换

2. 输入/输出地址空间(I/O)

32 位微处理器有两个独立的物理空间,一个是存储空间,这在前面已经介绍了;另一个是 I/O 空间,见图 2-10。

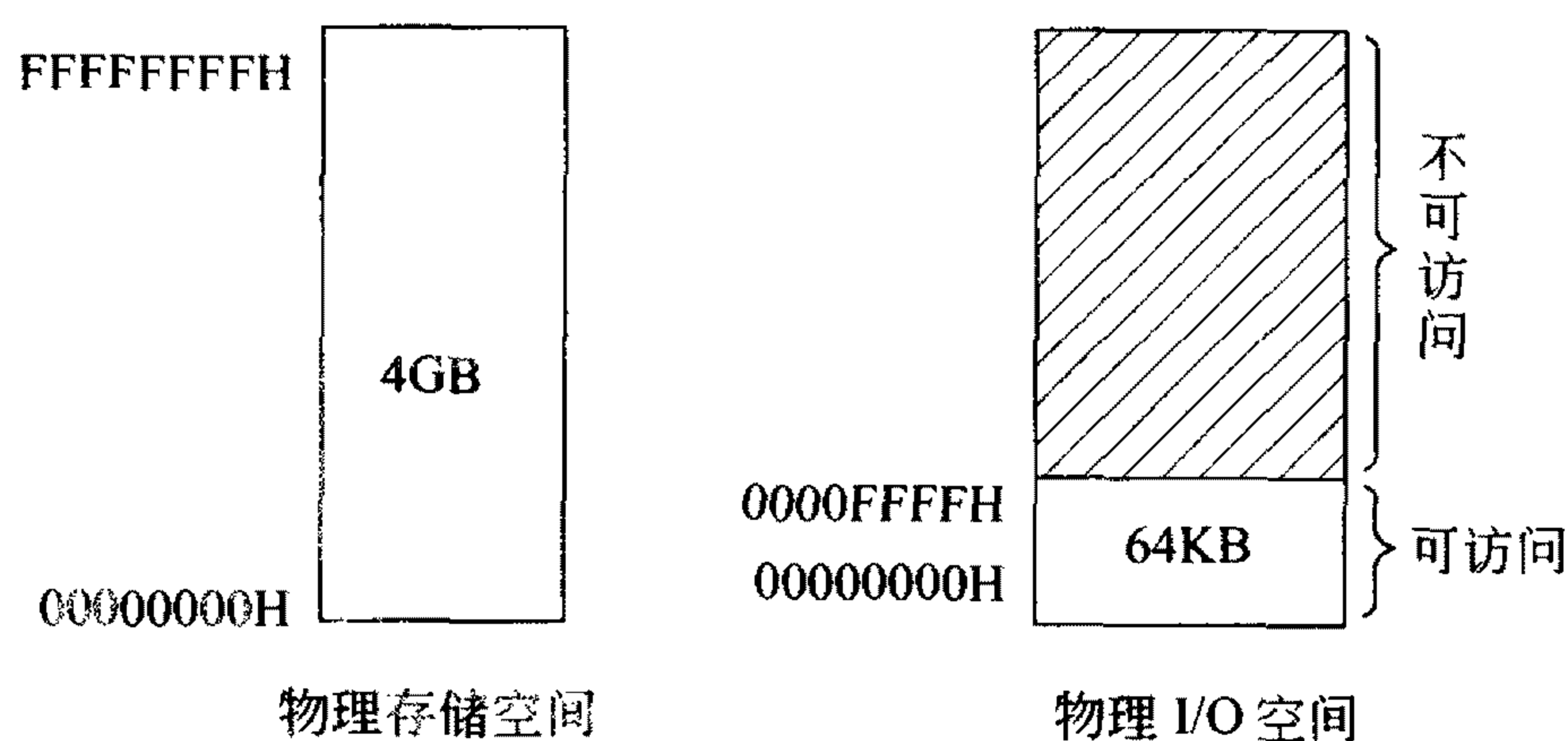


图 2-10 32 位微处理器的物理空间

80x86 的 I/O 空间由 2^{16} 个地址组成。它与存储地址不重叠,这是因为 80x86 微处理器芯片的 M/\overline{IO} 引脚把它们从逻辑上给区分开来了。

2.4.2 实地址模式

我们知道开发新一代微处理器时,都要考虑与前辈微处理器兼容的问题。在实地址模式下,32 位微处理器与它的前辈处理器 16 位的 8086 兼容,所以为 8086、80286 编写的程序不需要作任何修改,就可以在 32 位微处理器的实地址模式下运行,且速度更高。除

此之外,在实地址模式下,还能有效地使用 8086 所没有的寻址方式、32 位寄存器和大部分指令。在实地址模式下,32 位微处理器具有与 8086 同样的基本体系结构。归纳起来,实地址模式有如下几个特点:

- ① 寻址机构、存储器管理和中断机构均与 8086 一致。
- ② 操作数默认长度为 16 位,但允许访问 32 位寄存器组,在使用 32 位寄存器组时,指令中要加上前缀以表示越权存取。
- ③ 不用虚拟地址的概念,存储器容量最大为 1MB;采用分段方式,每个段大小固定为 64KB。
- ④ 实地址模式下,存储器中保留两个固定区域,一个为初始化区域,另一个为中断向量区。前者地址为 FFFF0H~FFFFFH,后者地址为 00000H~003FFH。

1. 存储空间及实地址模式下的存储器编址

实地址模式下只使用地址线的低 20 位,所以存储空间为 2^{20} 字节,即 1 兆字节。在实地址方式下分页功能是不允许的,所以线性地址就是物理地址。1MB 的内存单元的物理地址按照 00000H~FFFFFH 来进行编址。

由于实地址模式下使用的内部寄存器都是 16 位的,显然用单个寄存器不能给出 1MB 的内存单元所对应的物理地址,为此引入了分段概念。即一个段最多为 64KB,在通常的程序设计中,一个程序可以有代码段、数据段、堆栈段和附加数据段等,各段的段地址分别由代码段寄存器 CS、堆栈段寄存器 SS,数据段寄存器 DS、ES、FS 和 GS 这几个段寄存器给出。段寄存器都是 16 位的。

要计算一个存储单元的物理地址时,先要将它所在段的段寄存器的 16 位段地址值左移 4 位(相当于乘十进制数 16),得到一个 20 位的值,再加上 16 位的段内偏移量(该偏移量也称为有效地址),就形成了 20 位的物理地址。这样形成的物理地址特点是:所有的段总是起始于 16 字节的边界。图 2-11 所示的就是实地址模式下存储器单元 20 位物理地址的计算方法。

例:段寄存器 CS 内容为 1000H,偏移地址在 IP 寄存器中,为 8888H,实地址模式下的物理地址就是:

$$1000\text{H} \times 16 + 8888\text{H} = 18888\text{H}.$$

存储器中的操作数可以是 1 个字节,也可以是 2 个字节或 4 个字节。如果是多字节操作数,那么存放这个数的最低地址的单元所对应的 20 位物理地址就是该操作数的地址。

2. 保留的地址空间

在实地址方式下,有两个存储物理空间是需要保留的。地址 0000,0000H~0000,03FFH 是中断向量区,每一中断向量占用 4 个字节;地址 FFFF,FFF0H~FFFF,

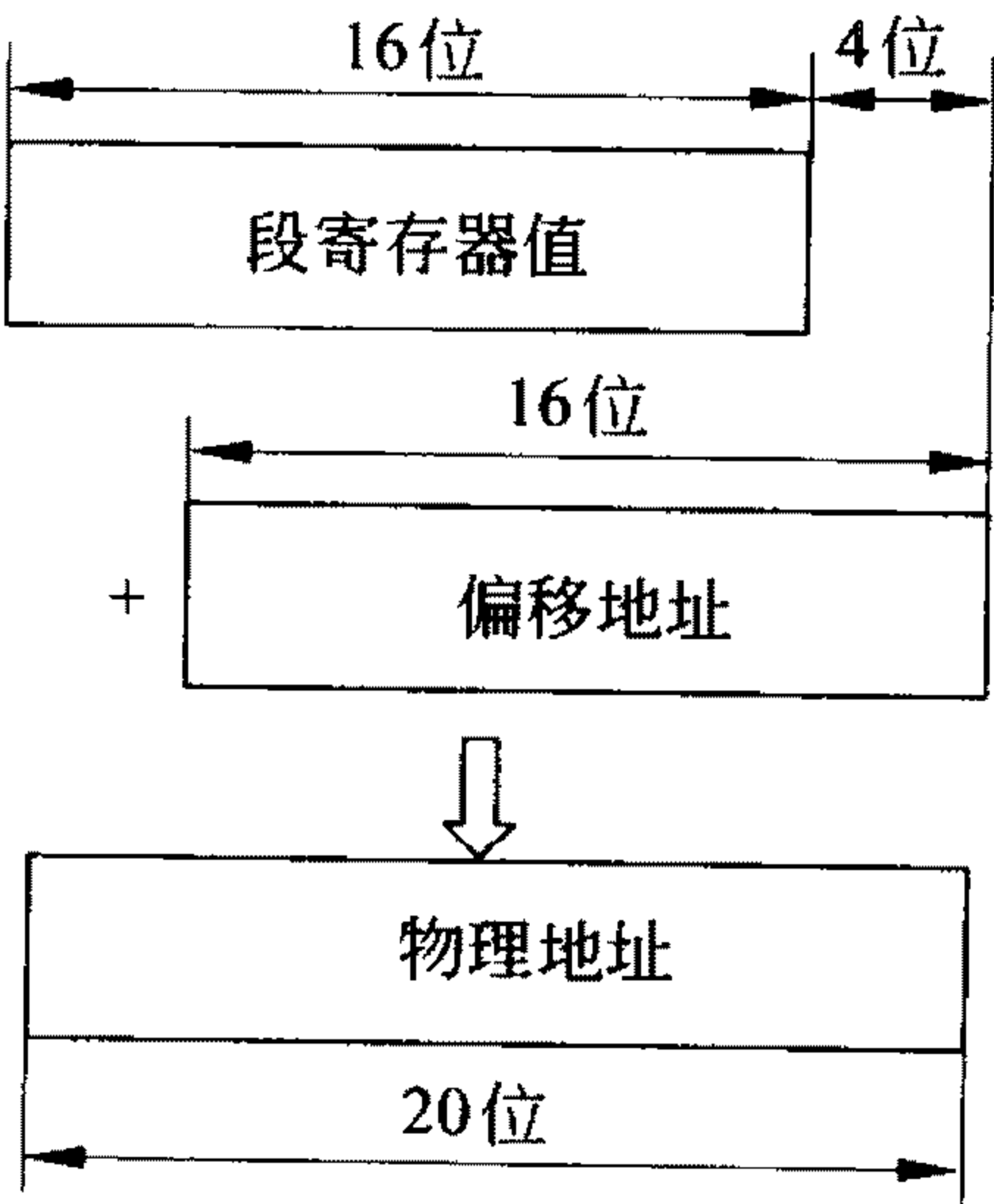


图 2-11 存储器单元 20 位物理地址的计算方法

FFFFH 为系统初始化区,当加电或复位时,物理地址自动置为 FFFF,FFF0H。

在实地址模式下,可以把 32 位微处理器的工作模式设置为保护模式。

2.4.3 保护虚拟地址模式介绍

保护虚拟地址模式又叫做保护模式,80386 以上处理器的存储管理及存储保护机制支持了保护虚拟地址模式。计算机软件由操作系统和应用程序组成,操作系统和应用程序之间有联系又互相独立。为了实现系统程序与应用程序之间、各个应用程序之间以及程序与数据之间互相独立所采取的措施叫做“保护”。这种保护机制是由硬件和软件共同配合完成的。32 位微处理器工作在保护模式时,充分发挥了 32 位微处理器所具有的存储管理功能以及硬件支撑的保护机制,这就为多用户操作系统的设计者提供了有力的支持,与此同时,在保护方式下,Pentium 微处理器也允许运行已有的 8086、80286、80386 和 80486 的软件,本小节仅就保护概念与特点、存储空间进行初步介绍,存储管理的具体实现在存储系统章节详细介绍。

1. 保护概念与保护模式特点

在程序运行过程中,应防止应用程序破坏系统程序、某一应用程序破坏了其他应用程序、错误地把数据当作程序运行等情况的出现。为避免这些情形所采取的措施称为保护。

32 位微处理器有多种保护方式,其中最突出的是采用了环保护方式。环保护是在用户程序与用户程序之间以及用户程序与操作系统之间实行隔离。32 位微处理器的环保护功能是通过设立特权级实现的,特权级分为 4 级(0~3),数值最低的特权级最高。在图 2-12 中,0 级被分配给操作系统的核心部分,如果操作系统被破坏了,整个计算机系统都会瘫痪,因此它所得到的保护级最高。

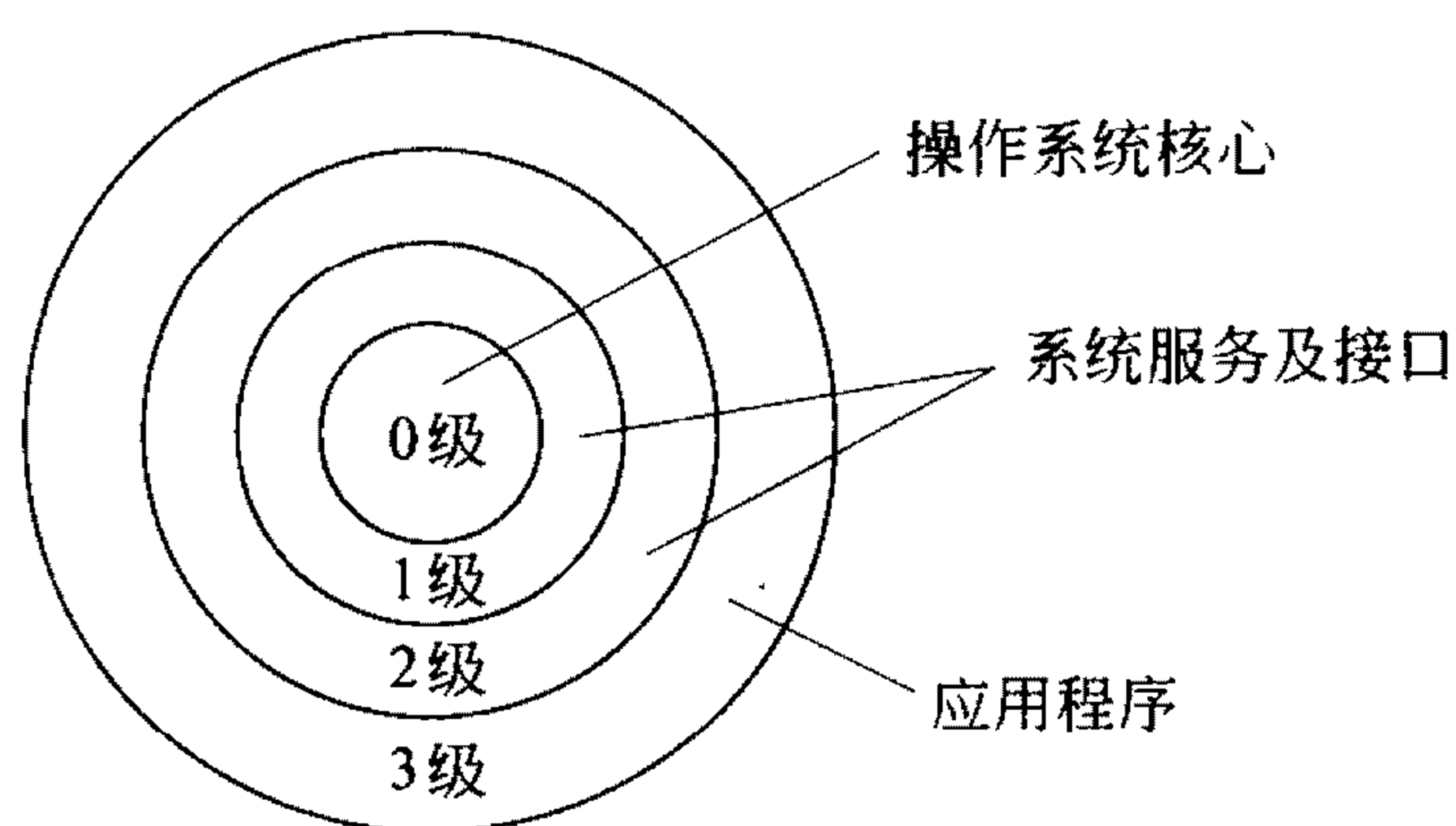


图 2-12 32 位微处理器的 4 级特权保护方式

32 位微处理器的特权规则有两条: 特权级 P 存储在某个段上的数据,只能由不低于 P 级的特权级进行访问;具有特权级 P 的程序或过程只能由在不高于 P 级上执行的任务调用。

综上所述,保护模式具有如下特点:

① 存储器用虚拟地址空间、线性地址空间和物理地址空间三种方式来进行描述,虚

拟地址就是逻辑地址。在保护模式下,寻址机构不同于 8086,需要通过一种称为描述符表的数据结构来实现对内存单元的访问。

② 程序员可以使用的存储空间称为逻辑地址空间,在保护模式下,借助于分段分页部件的功能将磁盘等存储设备有效映射到内存,使逻辑地址空间大大超过实际的物理地址空间,这样,主存储器容量似乎很大。

③ 既能进行 16 位运算,也能进行 32 位运算。

2. 存储空间

在保护方式下,32 位微处理器为每一个任务提供 2^{32} B(4GB)的物理空间,并允许程序在 2^{46} B(64TB)的逻辑空间运行。

2.4.4 虚拟 8086 模式介绍

32 位微处理器允许在实方式和保护方式下执行 8086 的应用程序。后者为系统设计人员提供了 32 位微处理器保护模式的全部功能,因而具有更大的灵活性。保护方式的功能之一是,能够在保护和多任务的环境中直接执行“实地址方式”的 8086 软件,这个特性称为“虚拟 8086 方式”。这不是一种实际的处理器方式,而是一种准操作方式。虚拟 8086 方式具有保护方式下的任务属性。

有了虚拟 8086 方式,32 位微处理器允许同时执行 8086 操作系统和 8086 应用程序以及 32 位操作系统和 32 位应用程序,因此在一台多用户的 32 位微处理器的计算机里,多个用户可以同时使用计算机。当操作系统或监控程序切换到虚拟 8086 方式时,处理器就模仿 Intel 8086 处理器来执行任务。8086 仿真状态下处理器的执行环境及扩展与实地址方式一样。这两种方式之间的主要区别在于,虚拟 8086 方式中,8086 程序以独立的保护方式任务运行。这样,8086 程序能够以 8086 的任务形式在吸取了保护方式优势的操作系统下运行,并可以使用保护方式机制,如使用保护方式存储管理机制、保护方式中断和异常处理机制以及保护方式多任务机制来为 8086 任务提供管理与保护。多任务机制允许多个虚拟 8086 方式任务与其他非虚拟 8086 方式任务一起在处理器上运行。

任何汇编或编译的在 Intel 8086 处理器上运行的新程序或老程序,都可以在虚拟 8086 方式任务上运行。使用处理器的多任务机制,8086 程序可以作为虚拟 8086 方式任务与普通保护方式任务一起运行。在保护方式下,可以通过软件切换到虚拟 8086 模式,虚拟 8086 模式具有如下特点:

① 可以执行 8086 的应用程序。

② 段寄存器的用法和实地址模式时一样,即段寄存器内容左移 4 位加上偏移地址为线性地址。

③ 存储器寻址空间为 1MB。

在虚拟 8086 方式下,还可以用与实地址方式相同的形式使用段寄存器,以形成线性基地址。通过使用分页功能,就可把虚拟 8086 方式下的 1MB 地址空间映射到 32 位微处理器的 4GB 的物理空间中的任何位置。

2.5 32 位微处理器的典型时序

2.5.1 时钟周期、总线周期和指令周期

任何计算机系统都具有时钟信号,它为系统工作提供了时序基准。微处理器内部的部件和子部件往往是以时钟信号作为启动条件。因此,计算机内部的时钟信号必须是一个有规律的脉冲信号。

时钟信号通常又被称为节拍脉冲,它的周期称为时钟周期(Clock Cycle)或 T 周期,是处理器处理操作的最基本单位。时钟周期是 CPU 的时间基准,它由计算机的主频决定。

若干个时钟周期可组成 1 个总线周期(Bus Cycle),所谓总线周期是指 CPU 从存储器或输入/输出端口存取 1 个字节或字所需要的时间。1 个总线周期通常由多个时钟周期组成,1 个时钟周期对应 1 个总线状态,状态(State)又称为 T。所以,1 个总线周期由多个 T 状态组成。8086 CPU 的总线周期至少由四个时钟周期组成,即有 4 个总线状态,分别以 T_1 、 T_2 、 T_3 和 T_4 表示。80486 CPU 的总线周期至少由两个时钟周期组成,分别以 T_1 和 T_2 表示。

一个总线周期完成一次数据传输,至少要有地址传送和数据传送两个过程。在第一个时钟周期 T_1 期间由 CPU 输出地址,在随后的 T 周期则完成数据的传输。换言之,数据传送必须在 $T_2 \sim T_4$ 内完成,否则,在最后一个 T 周期结束以后,会进入下一个总线周期。在实际应用中,当一些慢速设备在规定的几个 T 周期内无法完成数据读写时,那么就必须在总线周期中插入等待周期 T_w , T_w 也以时钟周期 T 为单位,但加入 T_w 的个数则与外部请求信号的持续时间长短有关。

CPU 每条指令的执行都由取指令、译码和执行等操作组成,CPU 读取并执行一条指令所花费的时间称为指令周期(Instruction Cycle),指令周期一般由若干个处理器周期组成。

2.5.2 Pentium 总线周期的时序分析

不同类型的 32 位微处理器,其总线周期也不相同。80486 微处理器的一般总线周期占用 2 个时钟的时间,即读或写都要 2 个时钟,这称为 2~2 周期。第一个 2 对应读,第二个 2 对应写。如果在读或写中增加了等待状态,则在读写的对应位置加上等待状态数。例如写操作需增加一个等待状态,则称为 2~3 周期。而 Pentium 微处理器的总线周期则更复杂。

1. Pentium 的总线状态

Pentium 有多种总线状态,各个状态之间是可以转换的。

① T_1 状态 这是总线周期的第 1 个时钟周期即第 1 个状态,此时,地址和状态信号

有效, \overline{ADS} 信号也有效, 同时, 外部电路可以将地址和状态送入锁存器。

② T_2 状态 此时数据出现在数据总线上, CPU 对 \overline{BRDY} 信号采样, 如 \overline{BRDY} 信号有效, 则确定当前周期为突发式总线周期, 否则为单数据传输的普通总线周期。

③ T_{12} 状态 这是流水线式总线周期中所特有的状态, 此时系统中有 2 个总线周期并行进行, 第 1 个总线周期进入 T_2 状态, 正在传输数据, 并且 CPU 采样 \overline{BRDY} 信号, 第 2 个总线周期进入 T_1 状态, 地址和状态信号有效, 并且 \overline{ADS} 信号也有效。

④ T_{2p} 状态 这是流水线式总线周期中所特有的状态, 此时系统中有 2 个总线周期, 第 1 个总线周期正在传输数据, 并且 CPU 对 \overline{BRDY} 采样, 但由于外设或存储器速度较慢, 所以, \overline{BRDY} 仍未有效, 也因此仍未结束总线周期, 第 2 个总线周期也进入第 2 个或后面的时钟周期。 T_{2p} 一般出现在外设或存储器速度较慢的情况下。

⑤ T_D 状态 这是 T_{12} 状态后出现的过渡状态, 一般出现在读写操作转换的情况下, 此时数据总线需要 1 个时钟周期进行过渡, 这种状态下, 数据总线上的数据还未有效, CPU 还未对 \overline{BRDY} 进行采样。

⑥ T_i 状态 这是空闲状态, 不在总线周期中, \overline{BOFF} 信号或 RESET 信号会使 CPU 进入此状态。

2. Pentium 的总线周期

Pentium 支持多种数据传输方式, 可以是单数据传输方式, 也可以是突发式传输方式。单数据传输时, 一次读写操作至少要用 2 个时钟周期, 可进行 32 位数据传输, 也可进行 64 位数据传输。突发式传输方式是 80486/Pentium 特有的一种新型传输方式, 用这种方式传输时, 在 1 个总线周期中可传输 256 位数据。与此相应, Pentium 的总线周期有多种类型。

按总线周期之间的组织方法来分, 有流水线和非流水线类型, 在流水线类型中, 前一个总线周期中已为下一个总线操作进行地址传输, 而在非流水线类型中, 每个总线周期独立进行一次完整的读操作或写操作, 与其他总线周期无关。

按总线周期本身的组织方法来分, 有突发式传输和非突发式传输类型。突发式传输时, 连续 4 组共 256 位数据可在 5 个时钟周期中完成传输, 这样可以加快对主存的信息存取。非突发式传输时, 通常用 2 个时钟周期构成一个总线周期传输单个数据, 可为 8 位、16 位、32 位或 64 位。

下面对常用的非流水线式读写周期的时序做一下说明。

这种总线周期至少占用 2 个时钟周期, 即 T_1 和 T_2 , 在外设或存储器较慢时, 则要多个 T_2 状态。在 T_1 状态, 段地址选通信号 \overline{ADS} 为低电平时, 在 ADDR 上地址有效, W/\overline{R} 如为低电平, 则进入读周期, 数据从外设或存储器送往 CPU。整个周期中, \overline{NA} 和 \overline{CACHE} 为高电平, 因此, 这是非流水线式的, 也不通过 Cache 进行读写。在 T_2 时钟周期, CPU 如采样到 \overline{BRDY} 信号为低电平, 说明外设已准备好, 于是 CPU 进行数据传输, 然后总线周期结束, 如果采样到的 \overline{BRDY} 仍为高电平, 则总线周期延长, 即在 T_2 状态等待, 直到 CPU 检测到 \overline{BRDY} 为低电平, 才结束总线周期。写操作时 W/\overline{R} 为高电平, 数据则来自 CPU, 其他信号和读操作时一样。图 2-13 是非流水线式读写周期的时序图。

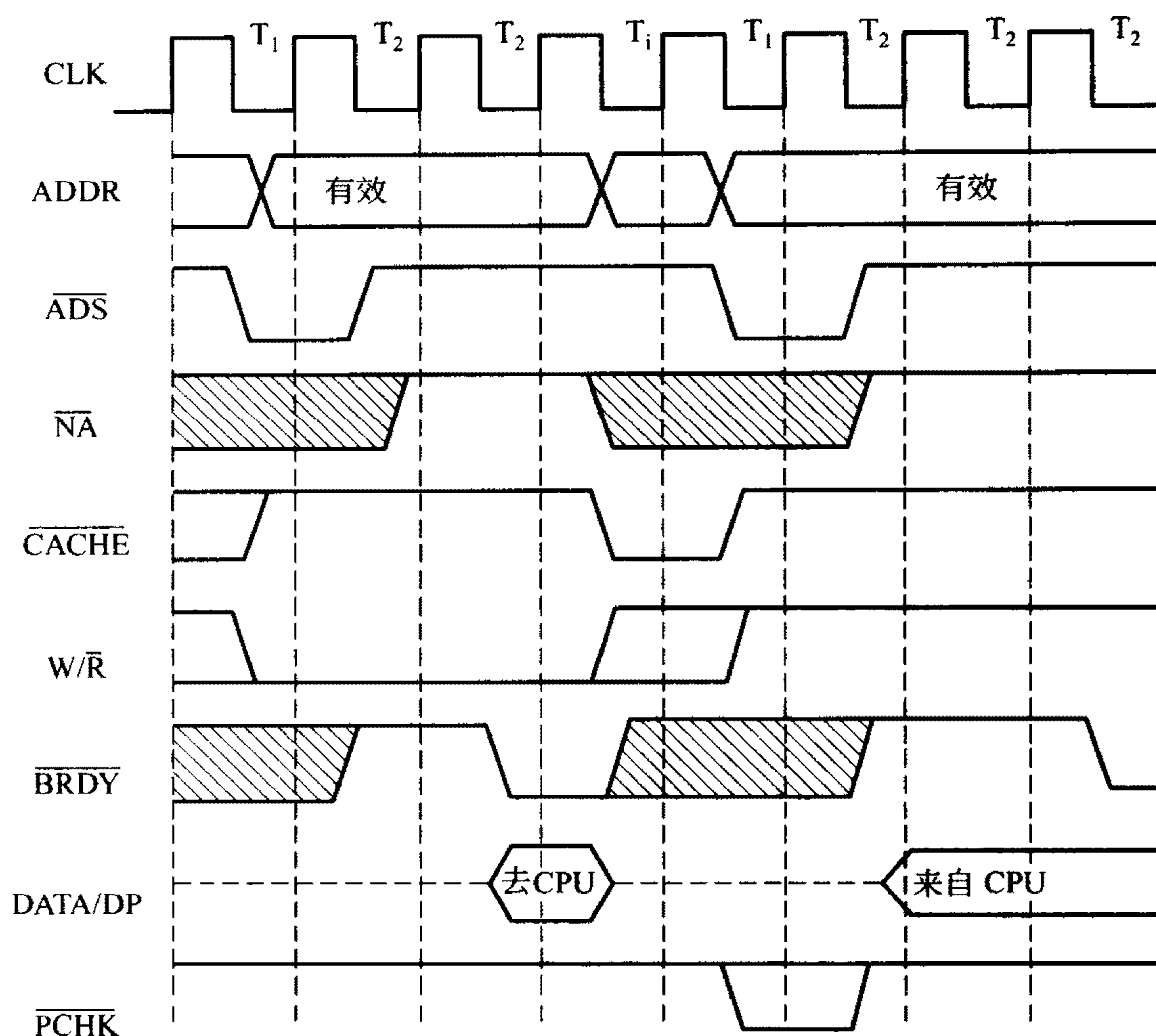


图 2-13 Pentium 非流水线式读写周期的时序图

习 题

1. Intel 微处理器的发展经历了哪几代?
2. 微处理器内部最基本的模块是什么? Pentium 微处理器内部有哪十大部件? 其基本结构特点是什么?
3. Pentium 微处理器有多少条数据线? 多少条地址线? 能访问的存储空间有多大?
4. 标志寄存器的功能及各种标志的含义是什么? 进位标志和溢出标志的区别是什么?
5. 在 32 位微处理器内部的通用寄存器中, 哪些可进行 32 位、16 位和 8 位的运算?
6. 在 32 位微处理器中, 哪些寄存器在应用程序中不可使用?
7. 32 位微处理器的工作模式有几种? 各自的特点是什么?
8. 32 位微处理器有哪三种存储地址空间? 32 位微处理器能访问的 I/O 空间是如何确定的?
9. 32 位微处理器工作在实地址模式时, 存储空间是多少? 20 位物理地址是如何形成的?
10. CS 的内容是 8000H, IP 的内容是 9832H, 求在实地址模式下的物理地址。
11. 系统复位后, EDX、CS 和 EIP 的内容是多少? 这些值表示什么含义?
12. 什么是时钟周期、总线周期和指令周期?

指令系统

3.1 概 述

3.1.1 指令的书写格式

指令是 CPU 执行某种操作的“命令”，CPU 全部指令的集合称为指令系统。

指令有两种书写格式：机器指令和符号指令。

机器指令用一串二进制数描述，计算机硬件只能识别、存储和运行机器指令。一般说，不同系列的 CPU 完成相同的操作，其机器指令描述是不相同的。机器指令不论在书写、阅读和记忆方面都十分困难，使用机器指令编程是不可想象的，为此引出了另一种表示方法，即符号指令。

符号指令是用规定的助记符和规定的书写格式书写的指令。

表 3-1 列举了 3 种操作，同时也写出了完成这些操作的符号指令和机器指令。

表 3-1 符号指令与机器指令对照表

操 作	80486 符号指令	80486 机器指令
1234H→AX	MOV AX,1234H	B8 34 12
AX+BX→AX	ADD AX,BX	03 C3
CX-DX→CX	SUB CX,DX	2B CA

其中，MOV、ADD、SUB 分别是传送指令、加法指令和减法指令的操作码助记符。助记符右侧用逗号间隔的是两个操作数，逗号左边为目标操作数，逗号右边是源操作数。MOV 指令的功能是把源操作数传送到目标寄存器中。ADD 指令的功能是完成两个操作数相加，结果写入目标寄存器。SUB 指令的功能是用目标操作数减源操作数，差值写入目标寄存器。很显然，符号指令比机器指令更容易理解，更容易记忆。

一条符号指令的机器指令有 1~16 个字节，它们在存储器中是连续存放的，CPU 规定：存放指令第一字节的内存地址称为指令地址。

3.1.2 符号指令的书写格式

符号指令的书写格式如下：



标号：操作码助记符 操作数助记符；注释

指令的核心要素是操作码和操作数。

标号代表该条指令的存放地址，它为程序分支、循环提供了转移目标。显然并非每一条指令都要设置标号，标号与符号指令之间用冒号做间隔符，标号的命名规则是：以字母或下划线开头，后跟字母、数字、下划线等，长度不超过 31 个字符。需要注意的是：指令的操作码助记符，伪指令助记符，CPU 寄存器的名称等“系统保留字”不能做为标号使用。

为了阅读方便，可以有注释，注释与符号指令用分号做间隔符，CPU 并不执行，只是在打印源程序清单时，照原样打印出来。

操作数是指令的操作对象，80486 指令操作数的长度可以是单字节、双字节或者四字节。多字节操作数是连续存放的，存放的规则是：低位字节存放在 i 单元中，高位字节存放在相邻的 $i+1$ 单元中，这一规则在以后的程序设计中十分有用，请读者务必注意。

用机器指令编写的程序称为目标程序，用符号指令设计的程序称为符号程序或汇编源程序，汇编源程序要经过编辑、汇编和链接才能生成 CPU 可执行的目标程序。

3.2 80486 寻址方式

前文已叙述过，指令由操作码和操作数两部分组成，操作数是指令的操作对象，在微型计算机系统中操作数有 3 种存放方式，即：

操作数就包含在本条指令当中，这种操作数，称为立即数。

操作数存放在 CPU 的某个寄存器中，这种操作数称为寄存器操作数。

操作数存放在存储器中，这种操作数，称为存储器操作数或内存操作数。

所谓寻址方式，就是在指令格式中用规定的助记符或者助记符表达式（即地址表达式）通知 CPU 怎样计算操作数的地址。

按粗线条划分，80486 有 7 种寻址方式，其中访问存储器有 5 种寻址方式。

3.2.1 立即寻址

立即寻址表示操作数包含在本条指令中，是指令的一部分，完整地取出该条指令之后也就获得了操作数。在符号指令中如何表示立即寻址？下面列举了几条指令，它们的功能是把源操作数（立即数）写入目标寄存器之中，其中源操作数部分，就是立即寻址。

```
MOV    AL,01010101B
MOV    BX,1234H
MOV    CL,4
MOV    DL,'A'
MOV    BL,0A6H
MOV    CX,3*5
MOV    EAX,12345678H
```

汇编语言规定：立即数必须以数字开头，以字母开头的十六进制数前面必须以数字0做前缀，数制用后缀表示，B表示二进制数，H表示十六进制数，D或者缺省表示十进制数，Q表示八进制数，程序员可以用自己习惯的计数制书写立即数。汇编程序在汇编时，对于不同进制的立即数一律汇编成等值的二进制数，用单引号括起来的字符汇编成相应的ASCII码，此外立即数还可以是用+、-、*、/表示的算术表达式，汇编程序按照先乘除后加减的规则自动计算，也可以用圆括号改变运算顺序。

3.2.2 寄存器寻址

如果操作数存放在CPU的某个寄存器中，如何通知CPU？在符号指令的操作数部分，写出寄存器的名称即可，下面列举的5条指令，其源操作数、目标操作数，均为寄存器寻址方式。

MOV	DS,AX	;AX寄存器的内容→DS
MOV	CL,BL	;BL寄存器的内容→CL
INC	SI	;SI寄存器的内容加1
DEC	DI	;DI寄存器的内容减1
ADD	EAX,EBX	;EAX,EBX的内容相加,结果→EAX

3.2.3 存储器操作数的寻址方式

本小节首先给出“逻辑地址”的概念，然后论述逻辑地址与物理地址的关系，最后介绍存储器操作数的5种寻址方式。

程序设计中最常见的是：操作数存放在某个逻辑段（例如数据段）的存储单元，或者运算结果要写入存储单元，显然CPU要取出（或写入）操作数必须事先知道存放操作数的那个单元的物理地址。由于80x86系列对存储器采用分段管理，所以在指令格式中直接写出存储单元的物理地址是不现实的，程序员只能给出存储单元的逻辑地址。逻辑地址经过汇编，由CPU内部的段页式管理部件最终生成物理地址，然后CPU再根据操作码的要求对该单元进行操作。

逻辑地址包含两部分，一是存储单元所在逻辑段的段基址，二是该单元的偏移地址。逻辑地址的一般书写格式为：

段寄存器名称：偏移地址表达式

其中，“段寄存器名称：”称为段超越前缀，它表示指令要访问的是哪一个逻辑段。

在实地址模式下，每个逻辑段首单元的物理地址是一个能被16整除的数，首单元物理地址除以16之后，其商值被称为该逻辑段的段基址。偏移地址是存储单元相对于段首单元的地址偏移量。例如：假设数据段首单元的物理地址为10000H，那么数据段的段基址就是1000H，物理地址为12345H的存储单元，其偏移地址就是2345H。逻辑地址“1000H：2345H”就代表物理地址为12345H的那个内存单元。

在实地址模式下，段寄存器的内容就是相关逻辑段的段基址，CPU对某一单元进行操作的时候，首先把该单元所在逻辑段的段寄存器内容乘以16然后加上该单元的偏移

地址就得到该单元的物理地址。偏移地址有多种生成方法,每一种方法对应着一种寻址方式,我们的学习重点就是深入理解每一种寻址方式的含义,并且按照汇编语言的要求,规范地写出每种寻址方式的地址表达式。下面详细介绍 80486 存储器操作数的 5 种寻址方式。

1. 直接寻址

直接寻址方式有两种书写格式。

① 在这种格式中,偏移地址表达式中直接写出存储单元的偏移地址,段超越前缀不能省略,否则将出现寻址错误。例如:

```
MOV    BX,DS:[1234H];取出数据段偏移地址为 1234H 的字单元的内容→BX
MOV    AL,ES:[2CH];取出 ES 附加段偏移地址为 2CH 的字节单元的内容→AL
```

这种书写格式,编程时很少使用,因为存储单元的分配是由 DOS 系统管理的,通常情况下程序员不知道要访问的那个存储单元的偏移地址是多少。

② 用变量名代表存储单元的偏移地址。

汇编语言规定,程序员可以为某个存储单元起一个名字,这个名字就称为存储单元的变量名。在一个源程序中,逻辑段之中或者逻辑段之间都不允许有重复定义的变量名,源程序经过汇编之后,存储单元的偏移地址就赋给了变量名,因此,在直接寻址方式中可以用变量名取代偏移地址表达式,又因为变量名是不允许重复定义的,所以段超越前缀可以省略。例如:取出数据段以 BUF 命名的双字单元的内容→EAX 寄存器,可以写成:

```
MOV    EAX,DS: BUF
```

或者写成:

```
MOV    EAX,BUF
```

2. 寄存器间接寻址

寄存器间接寻址(简称间接寻址或间址)是访问存储器最常用的寻址方式,这种寻址方式要求事先把存储单元的偏移地址写入规定的寄存器(为了描述方便,我们称它为间址寄存器)。指令的逻辑地址表达式部分用下列格式描述:

段寄存器:[间址寄存器]

对于约定的逻辑段其段超越前缀可以省略。例如:

假设数据段 BUF 字节单元有一个操作数 N,要求用间接寻址取出 N→AL 寄存器,在实地址模式下如何编程?

```
MOV    DS,数据段段基址
.....
MOV    BX,BUF 单元的偏移地址
MOV    AL,[BX]          ;访问数据段,用 BX 间址取数→AL
```

即:事先把 BUF 单元所在逻辑段的段基址→DS 寄存器,再把 BUF 单元的偏移地

址→BX 寄存器,做了这两项准备工作之后,CPU 在执行 MOV AL,[BX]时,首先把 DS 中的段基址乘 16,然后加上 BX 寄存器中的偏移地址从而得出 BUF 单元的物理地址,最后根据物理地址取出该单元的操作数 N→AL 寄存器。

从以上描述可以看出,BX 寄存器中存放的是 BUF 单元的偏移地址,而不是 BUF 单元中的操作数,换句话说,为了取出操作数 N,指令中没有给出偏移地址,而是给出了存放偏移地址的寄存器,这就是寄存器间接寻址的概念。哪些寄存器可以做间址寄存器呢? 80486 规定:

① 可以使用 BP,BX,SI,DI 4 个 16 位的寄存器做间接寻址寄存器,并且规定:使用 BP 寄存器间址,约定访问的是堆栈段,则段超越前缀“SS:”可省。使用 BX、SI、DI 寄存器间址,约定访问的是数据段,则段超越前缀“DS:”可省。

② 也可以使用 EBP、ESP 或者 EAX~EDX、ESI、EDI 这 8 个 32 位的寄存器做间接寻址寄存器,并且规定:使用 EBP、ESP 间接寻址,CPU 约定访问的是堆栈段,使用 EAX~EDX、ESI、EDI 间接寻址,CPU 约定访问的是数据段。例如:

```
MOV    BP,MESG 单元的偏移地址
MOV    CL,ES:[BP]           ;从 ES 附加段 MESG 字节单元取数→CL
MOV    SI,20H
MOV    EBX,[SI]             ;从数据段偏移地址 20H~23H 单元取数→EBX
```

3. 基址寻址

在这种方式中,存储单元的偏移地址为规定的基址寄存器的内容与一个常量(即位移量)之和,指令格式中,逻辑地址表达式写成:

段寄存器:[基址寄存器+位移量]

或者: 段寄存器:位移量[基址寄存器]

如果是访问约定的逻辑段,则段超越前缀可以省略。哪些寄存器可以做基址寄存器呢? 80486 规定:

① 可以用 BP、BX 这两个 16 位的寄存器做基址寄存器,使用 BP 进行基址寻址,约定访问的是堆栈段,使用 BX 进行基址寻址,约定访问的是数据段。

② 也可以使用 EBP、ESP、EAX~EDX、ESI、EDI 这 8 个 32 位的寄存器做基址寄存器,若使用 EBP、ESP 进行基址寻址约定访问的是堆栈段,若使用 EAX~EDX、ESI、EDI 进行基址寻址约定访问的是数据段,例如:

```
MOV    BP,BUF 单元的偏移地址
MOV    DL,DS:[BP+10]         ;访问数据段,从 BUF+10 字节单元
中取数→DL
MOV    EAX,NUM 单元的偏移地址
MOV    EDX,[EAX+10H]         ;从数据段 NUM+16~NUM+19 单
元取数→EDX
```

4. 变址寻址

变址寻址有两种格式:

① 有比例因子的变址寻址,在这种格式中:

存储单元的偏移地址 = 比例因子 × 变址寄存器的内容 + 位移量

指令格式中,完整的逻辑地址表达式写成如下形式:

段寄存器: [比例因子 × 变址寄存器 + 位移量]

或者: 段寄存器: 位移量 [比例因子 × 变址寄存器]

访问约定的逻辑段,段超越前缀可以省略。其中比例因子可以是 1、2、4、8 中的一个数,变址寄存器可以是 EBP 或者 EAX~EDX、ESI、EDI 这 7 个 32 位寄存器,并且规定:用 EBP 变址寻址,约定访问的是堆栈段,用 EAX~EDX、ESI、EDI 变址寻址,约定访问的是数据段。

② 没有比例因子的变址寻址,在这种格式中:

存储单元的偏移地址 = 变址寄存器的内容 + 位移量

指令格式中,完整的逻辑地址表达式写成如下形式:

段寄存器: [变址寄存器 + 位移量]

或者: 段寄存器: 位移量 [变址寄存器]

访问约定的逻辑段,段超越前缀可以省略。

在这种格式中,变址寄存器只能选择 SI、DI 这两个 16 位的寄存器,约定访问的是数据段。例如:

MOV AL, [2 * EBX + 10] ;从数据段偏移地址为 2 × EBX + 10 的单元中取数 → AL

MOV AH, [SI + 5] ;从数据段偏移地址为 SI + 5 的单元中取数 → AH

前者是有比例因子的变址寻址,后者是没有比例因子的变址寻址。

5. 基址加变址寻址

基址加变址寻址是基址和变址两种寻址方式的组合,它也有两种格式,区别仅在于地址表达式中是否含有比例因子。

① 有比例因子的基址加变址寻址,在这种格式中:

存储单元的偏移地址 = 基址寄存器内容 + 比例因子 × 变址寄存器的内容 + 位移量

指令格式中,完整的逻辑地址表达式写成如下形式:

段寄存器: [基址寄存器 + 比例因子 × 变址寄存器 + 位移量]

或者: 段寄存器: 位移量 [基址寄存器 + 比例因子 × 变址寄存器]

或者: 段寄存器: 位移量 [基址寄存器] [比例因子 × 变址寄存器]

访问约定的逻辑段,段超越前缀可以省略。

注意:在这种方式中基址寄存器和变址寄存器都必须是规定的 32 位寄存器。

② 没有比例因子的基址加变址寻址,在这种格式中:

存储单元的偏移地址 = 基址寄存器内容 + 变址寄存器的内容 + 位移量

指令格式中,完整的逻辑地址表达式写成如下形式:

段寄存器: [基址寄存器 + 变址寄存器 + 位移量]

或者: 段寄存器: 位移量 [基址寄存器 + 变址寄存器]

或者： 段寄存器：位移量[基址寄存器][变址寄存器]

访问约定的逻辑段,段超越前缀可以省略。在这种方式中基址寄存器和变址寄存器都必须是规定的 16 位寄存器。

【小结】

- ① 在基址、变址、基址加变址这 3 种寻址方式中,偏移地址表达式中的位移量是无符号整数。
- ② 带有比例因子的变址寻址,这种寻址方式常用于检索一维数组元素,当数组元素都是 2 字节长时,比例因子取 2,同理当一维数组元素都由 4 字节长或 8 字节长的元素组成时,比例因子应选取 4 或 8。
- ③ 带有比例因子的基址加变址,这种寻址方式常用于检索二维数组元素。
- ④ 在间址、基址、变址、基址加变址这 4 种寻址方式中,程序员可以使用 16 位的寄存器寻址,也可以使用 32 位的寄存器寻址,但请注意一个问题:当 CPU 工作在实地址模式的时候,段长度最大为 64KB,不论你采用 16 位寄存器寻址还是 32 位寄存器寻址都必须保证 CPU 最终算出的偏移地址不超过 FFFFH,而且操作数最高字节单元的偏移地址也不能超过 FFFFH,否则执行寻址操作时系统将要瘫痪! 例如:

```

执行:      MOV EBX,10000H      ;EBX 中偏移地址大于 FFFFH
           MOV AL, [EBX]        ;执行该指令,系统瘫痪

执行:      MOV SI, 0FFFFH      ;虽然 SI 中偏移地址不大于 FFFFH
           MOV AX, [SI]        ;但[SI]寻址的是双字节数,高字节偏移地址为
                               ;10000H,超出了 FFFFH,执行该指令系统死机。
    
```

3.2.4 80486 寻址方式的段约定和段超越

表 3-2 给出了存储器寻址时规定使用的寄存器,以及约定访问的逻辑段,由于基址加变址寻址是基址寻址和变址寻址的组合,所以表中没有列出基址加变址寻址时规定使用的寄存器。

表 3-2 存储器寻址规定使用的寄存器与段约定

间接寻址寄存器	基址寻址寄存器	变址寻址寄存器	约定访问的逻辑段
BP	BP		堆栈段
BX,SI,DI	BX	SI,DI	数据段
EBP,ESP	EBP,ESP	EBP	堆栈段
EAX,EBX,ECX	EAX,EBX,ECX	EAX,EBX,ECX	数据段
EDX,ESI,EDI	EDX,ESI,EDI	EDX, ESI,EDI	

表中指出:在用间址、基址、变址、基址加变址访问存储器时,如果使用 BP 或者 EBP、ESP 参与寻址,则 CPU 自动认为这是访问堆栈段,为此在指令的逻辑地址表达式部分,段超越前缀“SS:”可以省略。反之,如果使用 BP 或者 EBP,ESP 参与寻址,但程序员真正想访问的是堆栈段之外的其他逻辑段,那么逻辑地址表达式中必须明确写出相关

逻辑段的段超越前缀(否则将出现寻址错误),为了简单起见,以间址为例:

MOV	AL,DS:[BP]	;用 BP 间址访问数据段
MOV	AL,ES:[BP]	;用 BP 间址访问 ES 附加段
MOV	AL,FS:[EBP]	;用 EBP 间址访问 FS 附加段

表中还指出,间址寻址使用 BX、SI、DI,基址寻址使用 BX,变址寻址使用 SI、DI,以及使用 EAX~EDX、ESI、EDI 参与寻址,CPU 自动认为这是访问数据段,所以指令的逻辑地址表达式部分,段超越前缀“DS:”可以省略,反之,如果用这些寄存器参与寻址数据段以外的其他逻辑段,必须明确写出相关逻辑段的段超越前缀,例如:

MOV	AL,CS: [BX]	;用 BX 间址访问代码段
MOV	AL,ES: [SI+5]	;用 SI 变址,访问 ES 附加段
MOV	AL,GS: [EAX+10]	;用 EAX 基址寻址,访问 GS 附加段

由于源程序中,不允许出现重复定义的变量名,因此使用变量名直接寻址访问存储器,不需要另加段超越前缀。

此外,执行堆栈操作指令(PUSH,POP,RET…),CPU 自动寻址堆栈段,并且自动使用 SP(或 ESP)的当前值做为偏移地址完成压入或弹出操作,程序员无法用段超越前缀进行干预。

最后要指出的是,绝大多数程序都使用段约定访问存储器,读者应努力使自己习惯这种编程风格,提高程序的可读性。

3.3 80486 标志寄存器

标志寄存器含有两类标志,即状态标志和控制标志。状态标志记录了当前指令执行后的一些状态信息,控制标志控制微处理器操作。

80486 标志寄存器为 32 位结构,实际使用 15 位,共 14 个标志。图 3-1 为 80486 标志寄存器的位结构。

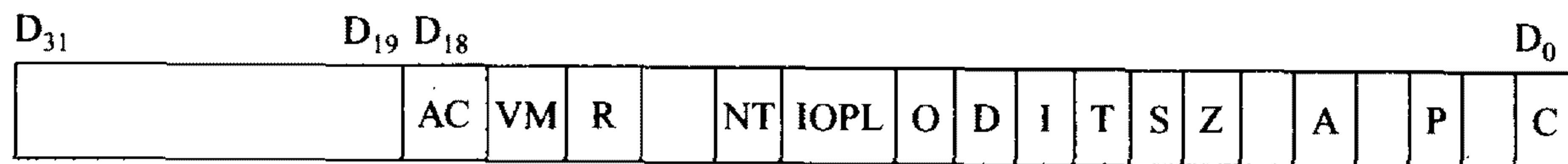


图 3-1 80486 SX 标志寄存器

下面分别介绍各标志位的功能,以及在一般情况下标志位的置 0/置 1 条件:

1. C 标志(进位/借位标志)

C 标志记录了加/减运算之后,最高位的进位值/借位值。

字节加/减、字加/减、双字加/减时,最高位产生进位/借位,则 C 标志置 1,否则 C 标志置 0,移位和循环移位指令也影响 C 标志。

2. A 标志(辅助进位/借位标志)

A 标志又称半进位/半借位标志。

加/减运算时, D_3 位向 D_4 位有进位/借位, 则 A 标志置 1, 否则置 0。

3. S 标志(符号标志)

S 标志记录了运算结果的最高位的值。字节运算后结果的 D_7 位为 1, 字运算后结果的 D_{15} 位为 1, 双字运算后结果的 D_{31} 位为 1 均使 S 标志置 1, 否则 S 标志置 0。如果是有符号数运算, S 标志为 0 表示结果为正数, S 标志为 1, 表示结果为负数。

4. Z 标志(全零标志)

当运算结果为全零时 Z 标志置 1, 否则置 0。

5. P 标志(奇偶性标志)

运算结果的低 8 位中, 1 的个数为偶数个则 P 标志置 1, 否则 P 标志置 0。在数据通信中, 该标志是校验数据传送正确性的一种手段。

6. O 标志(溢出标志)

运算结果产生溢出则 O 标志置 1, 否则 O 标志置 0。

溢出标志是很重要的标志, 只有正确理解有关溢出的基本概念, 才能正确地使用溢出标志。

什么是溢出?

运算结果超出了目标寄存器或目标单元所能表示的范围就称为溢出, 例如运算结果出现以下情况之一, 都称为溢出。

① 两个 n 位的无符号二进制数相加, 结果大于 $2^n - 1$;

② 两个 n 位的有符号二进制数相加, 结果大于 $2^{n-1} - 1$ 或小于 -2^{n-1} , 其中 n 为字长, 即 n 为 8、16 或 32。

从以上分析可以看出, 溢出和操作数的性质有关。但是操作数的性质是由程序员定义的, 计算机硬件无法知道参与运算的操作数是有符号数还是无符号数, 因此硬件只能默认一种选择, 即参与运算的操作数都是有符号数。做了这样的硬性规定之后, 当两数的符号位相同而且和结果的符号位相异时, 硬件就自动使 O 标志置 1, 否则使 O 标志置 0。

搞清楚溢出标志的置位条件之后, 程序员怎样判断溢出呢? 在程序设计时, 如果程序员定义操作数是有符号数, 则运算之后应测试 O 标志, O 标志为 1 表示溢出, 如果程序员定义操作数是无符号数, 则运算之后应测试 C 标志, C 标志为 1 表示溢出。

7. D 标志(方向标志)

执行 STD 指令则 D 标志置 1, 执行 CLD 指令 D 标志置 0。方向标志的作用是: CPU

在执行串操作指令时,控制字符串指针的调整方向,D标志为0,进行增址型调整,D标志为1,进行减址型调整。

8. I 标志(中断允许标志)

I标志控制CPU是否响应来自引脚INTR的可屏蔽中断请求。执行STI指令I标志置1,CPU响应可屏蔽中断;执行CLI指令I标志置0,CPU不响应可屏蔽中断。

9. T 标志(陷阱标志)

该标志置1后,CPU将进入单步执行方式,即每执行一条指令后都产生一次“单步中断”,若该标志为0则CPU连续执行指令。指令系统中没有设置使T标志置0/置1的指令,需要时可编写一段程序使T标志置0/置1。

10. IOPL 标志(I/O 特权级标志)

该标志占两位,表示0级~3级4个I/O特权级,0为最高级,3为最低级,该标志用于保护模式下的输入/输出操作,只有当任务的特权级高于或等于IOPL时,执行I/O指令才能保证不产生异常。

11. NT 标志(任务嵌套标志)

该标志仅用于保护模式。在保护模式下,如果当前执行的A任务是嵌套于B任务之中的,则NT标志置1,从而指示CPU A任务执行完毕要返回到B任务之中。

12. R 标志(恢复标志)

该标志与调试寄存器配合使用,当CPU响应“断点异常中断”时R标志置1,然后标志寄存器压栈再转入相应的断点处理程序,此时若遇到调试故障也不再产生异常中断,断点处理程序结束后返回断点指令。

13. VM 标志(虚拟标志)

如果CPU工作在保护模式而VM又被置成1,则CPU就转换成虚拟8086操作模式。

14. AC 标志(对准检查标志)

若AC标志为1,且CR₀寄存器的AM位也为1,则进行字、双字或4字的对准检查,若要访问的内存操作数未按边界对准,则发生异常中断。什么是边界?CPU规定,访问字操作数应从偶地址开始,访问双字操作数应从4的整数倍地址开始,访问4字操作数应从8的整数倍地址开始,不符合上述规定就是越界。

在上述14个标志当中,C、P、A、Z、S、O为状态标志,它们记录了当前指令执行后的一些特征信息(并非所有指令都对它们产生影响),为程序转移提供测试条件;D标志是控制标志,用于串操作指令中控制字符串指针的调整方向;其余标志均属于系统标志。

前 9 种是 8086/8088 的标志位, IOPL、NT 是 80286 开始增加的标志位; R、VM 是 80386 开始增加的标志位; AC 为 80486SX 增加的标志位。

3.4 80486 基本集指令

80486 指令系统是在 8086/8088、80286、80386 指令基础上发展形成的。和 80286 比较, 增加了 32 位操作和访问存储器的 32 位寻址方式。

80486 可以工作在实地址模式、保护模式和虚拟 8086 模式, 为了支持系统工作模式, 指令系统中设计了系统管理指令、保护模式控制指令以及高级语言支持指令……作为汇编语言程序设计的基础。我们仅介绍 80486 的基本集指令, 有关中断和输入/输出的指令在相关章节中介绍。

在介绍指令的时候, 没有刻意指出哪些指令或者指令的哪一种格式是 80486 新增加的, 这样做的目的是减轻学习负担, 另外, 在介绍指令的过程中, 需要适当地穿插一些指令的运用和编程, 这样就必然涉及汇编语言的若干基本规定, 为此请读者提前阅读 4.3 节宏汇编基本语法和 4.4 节数据定义伪指令。

3.4.1 传送类指令

传送类指令执行后, 源操作数不变, 不影响状态标志。(标志寄存器传送指令除外) 这类指令又可以分为: 通用传送指令、堆栈操作指令和输入/输出指令三类。其中输入/输出指令在第 8 章介绍。

1. 通用传送指令

(1) 数据传送指令

格式: MOV 目标操作数, 源操作数

功能: 把源操作数传送到目标寄存器或目标单元, 源操作数不变, 不影响状态标志。

说明:

① 源操作数可以是 8 位、16 位或 32 位的立即数、寄存器、段寄存器或内存操作数。目标是与源等长的寄存器、段寄存器(CS 除外)或内存单元。源、目标不能同为内存操作数。

② 源、目标操作数类型必须匹配, 如果目标是用间址、基址、变址或基址加变址寻址的内存单元, 而源是单字节或双字节立即数, 则必须用 PTR 运算符说明目标操作数的属性, 例如:

```
MOV    BYTE PTR [BX], 12H      ;12H→BX 间址的字节型单元
MOV    WORD PTR [SI+5], 1234H  ;1234H→SI 变址的字型单元
```

③ 不能向段寄存器写入立即数, 对段寄存器初始化应借用一个 16 位的寄存器过渡, 例如:

```
MOV    AX,DATA           ;DATA 为数据段段名,汇编后即为 DATA 段的段基址
MOV    DS,AX
```

④ 以 CS 为目标的一切传送指令都是非法的。

(2) 符号扩展传送指令

格式: MOVSX 目标寄存器,源操作数

功能与说明: 目标为 16 或 32 位的寄存器,源是小于(或等于)目标字长的寄存器或内存操作数。

该指令将源操作数的符号位向高位扩展使其与目标字长相同,然后再传送到目标寄存器,而源操作数不变,例如:

```
MOV     DL,-16           ;DL=F0H
MOVSX   BX,DL           ;BX=FFF0H,而 DH,DL 不变
```

(3) 零扩展传送指令

格式: MOVZX 目标寄存器,源操作数

功能: 与 MOVSX 类似,只是将源操作数高位用零补足 16 位或 32 位,然后再传送到目标寄存器。

(4) 偏移地址传送指令

格式: LEA 目标寄存器,源操作数

功能与说明: 目标为 16 位或 32 位的寄存器,源操作数为内存操作数。该指令将内存单元的偏移地址(而不是该单元的内容)传送到目标寄存器中,例如:

```
LEA     BX,BUF           ;将 BUF 单元的偏移地址→BX
LEA     EAX,[SI+5]       ;将数据段采用 SI+5 变址寻址的那个单元的偏移地址→EAX
```

偏移地址传送还可以用 MOV 指令完成,例如:

```
MOV     BX,OFFSET BUF ;BUF 单元的偏移地址→BX
```

OFFSET 是汇编语言提供的运算符,在源程序汇编时完成偏移地址计算,执行该指令时完成赋值。

(5) 指针传送指令

格式: 操作码助记符 目标寄存器,源操作数

功能与说明:

① 操作码助记符有: LDS、LES、LFS、LGS、LSS,其后两位字母代表段寄存器,它们是隐含的目标寄存器,共有 5 条地址指针传送指令。

② 如果目标是 16 位通用寄存器,则源操作数应是 32 位内存操作数,指令执行后,内存操作数高 16 位→隐含的段寄存器,低 16 位→目标指定的通用寄存器。

③ 如果目标是 32 位通用寄存器,则源操作数应是 48 位内存操作数,指令执行后,内存操作数高 16 位→操作码指定的段寄存器,低 32 位→目标指定的通用寄存器。例如:

```
设数据段: ADDR1    DF    1234567890ABH
           ADDR2    DD    1A2B3C4DH
```


代码段：对 DS 初始化

```
LES    EBX, ADDR1    ;ES=1234H,EBX=567890ABH
LDS    SI, ADDR2     ;DS=1A2BH,SI=3C4DH
```

(6) 标志寄存器传送指令

格式：LAHF
SAHF

功能：LAHF 将标志寄存器低 8 位传送到 AH 寄存器中,SAHF 将 AH 寄存器的内容传送到标志寄存器低 8 位。

(7) 交换指令

格式：XCHG 目标操作数,源操作数

功能与说明：源和目标等长的寄存器操作数、内存操作数,但不能同为内存操作数,该指令完成源、目标操作数互换。

(8) 字节交换指令

格式：BSWAP 32 位寄存器

功能：将 32 位通用寄存器的 31 位~24 位与 7 位~0 位交换,23 位~16 位与 15 位~8 位交换。

(9) 查表指令

格式：XLAT 表头变量名

功能：取出 DS:[BX+AL]中的一个字节→AL,或者取出 DS:[EBX+AL]中的一个字节→AL。

说明：若干单字节数的集合称为字节表,存放第一个表项的内存变量名称为“表头”,该指令查找存放在数据段中的字节表。指令执行前应做两项准备工作：表头偏移地址送 BX 或 EBX;要查找的表项相对于表头的地址偏移量送 AL,则指令执行时 CPU 以 BX 或 EBX 为基址寄存器,以 AL 的值为位移量进行基址寻址取出相应的表元素送 AL。

设某数码管显示电路,其八段数码管的字形编码如表 3-3 所示,要求查找与 NUM 单元的数对应的字形编码。

表 3-3 数码管字形编码表

字形	0	1	2	3	4	5	6	7	8	9	A	b	C	d	E	F
编码(H)	3F	06	5B	4F	66	6D	7D	07	7F	6F	77	7C	39	5E	79	71

首先应在数据段按字形 0~F 的顺序设置一张字形编码表：

```
TAB    DB    3FH,06H,5BH,4FH,66H,6DH,7DH,07H
        DB    7FH,6FH,77H,7CH,39H,5EH,79H,71H
NUM    DB    ××      ;0~15 中的任一数
```

代码段设置如下指令,即可查出和 NUM 单元数对应的字形编码：

.....

```

MOV    BX,OFFSET TAB
MOV    AL,NUM
XLAT   TAB           ;AL=相应的字形编码

```

2. 堆栈操作指令

堆栈是人为定义的一块连续的内存空间,用来暂存数据。这些数据,按照“先进后出”的规律存取。在 80x86 系列中,栈底为堆栈空间的高地址单元,栈顶为低地址单元。数据进栈后,栈顶向低地址方向浮动,数据出栈后,栈顶向高地址方向调整。一个 16 位或 32 位的数据的进栈规律是:高位字节存入高地址单元,低位字节存入低地址单元。一个 16 位或 32 位数据的出栈规律是:低位字节弹出到目标操作数低位,高位字节弹出到目标操作数高位。为了指示栈顶的当前位置,用 SP 或 ESP 存放栈顶的偏移地址。

(1) 进栈指令

格式: PUSH 源操作数

说明: 进栈数据可以是段寄存器内容,也可以是 16 位或 32 位的立即数、通用寄存器或内存操作数。如果内存操作数不是采用直接寻址,则必须用 PTR 运算符说明其属性。

功能: 执行时,首先调整堆栈指针,然后把源操作数压栈。即:

16 位操作数进栈: $SP-2 \rightarrow SP$, 16 位操作数 $\rightarrow SS: [SP]$ 的 2 个单元;

32 位操作数进栈: $ESP-4 \rightarrow ESP$, 32 位操作数 $\rightarrow SS: [ESP]$ 的 4 个单元。

如果源操作数是 SP 或 ESP,则 80486CPU 将调整前的 SP 或 ESP 压栈。例如:

```

PUSH   AX                ;SP-2→SP, AX→SS: [SP]的 2 个单元
PUSH   WORD PTR [BX]     ;SP-2→SP, DS: [BX]的一个字→SS: [SP]的 2 个单元
PUSH   DWORD PTR [SI+5]  ;ESP-4→ESP,
                        ;DS: [SI+5]的一个双字→SS: [ESP]的 4 个单元

```

(2) 出栈指令

格式: POP 目标操作数

说明: 目标操作数可以是除 CS 之外的段寄存器,也可以是 16 位或 32 位的通用寄存器、内存单元。如果内存单元不是采用直接寻址则必须用 PTR 运算符说明其属性。

功能: 先从栈顶弹出 2 个或 4 个字节送目标操作数,然后调整堆栈指针。即:

16 位操作数出栈: $SS: [SP]$ 的 2 个字节 \rightarrow 16 位目标操作数, $SP+2 \rightarrow SP$,

32 位操作数出栈: $SS: [ESP]$ 的 4 个字节 \rightarrow 32 位目标操作数, $ESP+4 \rightarrow ESP$ 。

```

例如:  POP    BX                ;SS: [SP]的 2 个字节→ BX, SP+2→SP
        POP    DWORD PTR [SI]  ;SS: [ESP]的 4 个字节→DS: [SI]的 4 个单元
                        ;ESP+4→ESP

```

设: $SS=0200H$, $SP=0012H$, $AX=1234H$, $CX=5678H$, 执行以下 3 条指令后,堆栈空间的数据变化如图 3-2 所示。

```

PUSH    AX
PUSH    CX
POP     BX

```

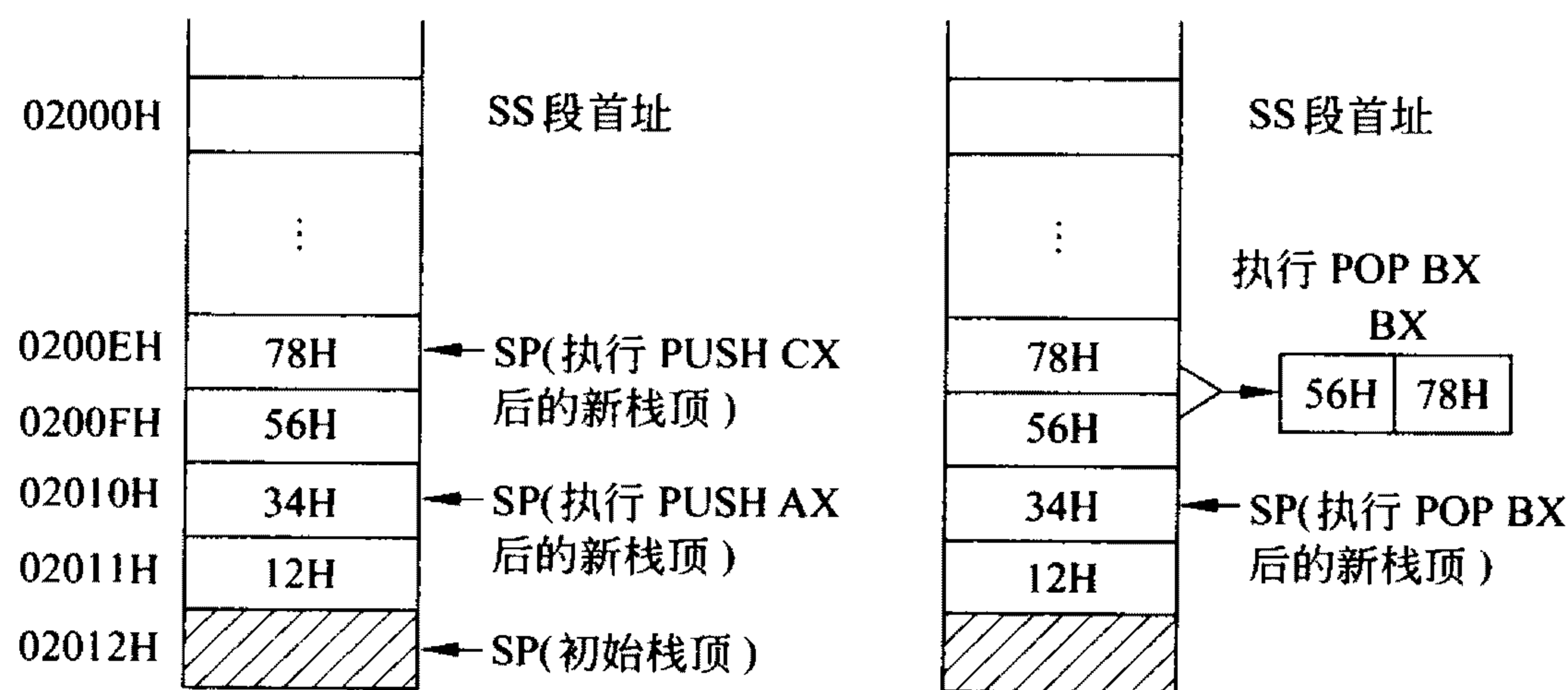


图 3-2 堆栈空间示意图

(3) 16 位标志寄存器进栈/出栈指令

格式: PUSHF

POPF

功能: 执行 PUSHF 时, 首先 $SP-2 \rightarrow SP$, 然后标志寄存器低 16 位 $\rightarrow SS:[SP]$ 的两个单元。执行 POPF 时, 先从栈顶弹出 2 个字节 \rightarrow 标志寄存器低 16 位, 然后 $SP+2 \rightarrow SP$ 。

(4) 32 位标志寄存器进栈/出栈指令

格式: PUSHFD

POPFD

功能: 执行 PUSHFD 时, 首先 $ESP-4 \rightarrow ESP$, 然后 32 位标志寄存器 $\rightarrow SS:[ESP]$ 的 4 个单元。执行 POPFD 时, 先从栈顶弹出 4 个字节 \rightarrow 32 位标志寄存器, 然后, $ESP+4 \rightarrow ESP$ 。

(5) 16 位通用寄存器进栈/出栈指令

格式: PUSHA

POPA

功能: 执行 PUSHA 时, 首先 $SP-16 \rightarrow SP$, 然后依次把 AX、CX、DX、BX、SP、BP、SI、DI 的内容压入堆栈, 进栈的 SP 值是调整前的值。执行 POPA 时, 先从栈顶弹出 16 个字节, 并依次装入 DI、SI、BP、SP、BX、DX、CX、AX。

(6) 32 位通用寄存器进栈/出栈指令

格式: PUSHAD

POPAD

功能: 执行 PUSHAD 时, 首先 $ESP-32 \rightarrow ESP$, 然后依次把 EAX、ECX、EDX、EBX、ESP、EBP、ESI、EDI 的内容压入堆栈, 进栈的 ESP 值是调整前的值。执行 POPAD 时, 先从栈顶弹出 32 个字节 (8 个双字), 并依次装入 EDI、ESI、EBP、ESP、EBX、EDX、ECX、EAX。

3.4.2 算术运算指令

80486 提供了一套二进制数的加、减、乘、除运算指令, 其运算对象可以是 8 位、16

位、32 位的有符号数或无符号数,另外还提供了若干调整指令,使得运算对象可以是组合的 BCD 码数(即压缩的 BCD 码数)或非组合的 BCD 码数(即非压缩的 BCD 码数)。这类指令执行后要影响标志寄存器中的状态标志。

1. 基本四则运算

(1) 二进制数加法指令(Add)

格式: ADD 目标操作数,源操作数

功能: 源操作数+目标操作数→目标操作数。

(2) 二进制数减法指令(Subtract)

格式: SUB 目标操作数,源操作数

功能: 目标操作数-源操作数→目标操作数。

(3) 二进制数加进位指令(Add wath carry)

格式: ADC 目标操作数,源操作数

功能: 源操作数+目标操作数+C 标→目标操作数。

其中的 C 标是上一条指令执行后产生的 C 标志。

(4) 二进制数减进位指令(Subtract wath borrow)

格式: SBB 目标操作数,源操作数

功能: 目标操作数-源操作数-C 标→目标操作数。

其中的 C 标是上一条指令执行后产生的 C 标志。

说明:

① 上述 4 条指令,目标操作数为 8 位、16 位或 32 位的寄存器操作数、内存操作数,源操作数是与目标等长的立即数、寄存器操作数或内存操作数,但源、目标不可同为内存操作数。

② 如果源操作数是单字节或双字节立即数,而目标是用间址、基址、变址、基址加变址寻址的内存操作数,则目标操作数必须用 PTR 运算符说明是字节型还是字型,否则汇编时出错。例如:

ADD	[BX],12H	;错误
ADD	BYTE PTR [BX],12H	;正确
SUB	[SI+2],1234H	;错误
SUB	WORD PTR [SI+2],1234H	;正确

③ 指令执行后,影响 A、C、O、P、S、Z 6 个标志,例如:实现下列多字节二进制数的加法运算:

5	6	7	8	8	7	8	5	H	
+	7	8	9	A	8	7	8	5	H
<hr/>									
C	F	1	3	0	F	0	A	H	

运算应从最低位开始,当低位向高位有进位时,高位要进行“加进位”操作,编程时,如果每次运算均以字节为单位,共需进行 4 次,如果以字为单位只进行 2 次运算。本例用 32 位二进制数相加,只进行 1 次运算。

首先在数据段设置两个多字节数：

```
FIRST      DD  56788785H
SECOND     DD  789A8785H
SUM        DD  ?
```

解 1 采用直接寻址，字节运算共需 12 条指令。

```
MOV  AL,BYTE PTR FIRST
ADD  AL,BYTE PTR SECOND      ;进位值→C 标
MOV  BYTE PTR SUM,AL
MOV  AL,BYTE PTR FIRST+1
ADC  AL,BYTE PTR SECOND+1    ;加进位
MOV  BYTE PTR SUM+1,AL
.....
```

解 2 采用直接寻址，字运算只需 6 条指令。

```
MOV  AX,WORD PTR FIRST
ADD  AX,WORD PTR SECOND
MOV  WORD PTR SUM,AX
MOV  AX,WORD PTR FIRST+2
ADC  AX,WORD PTR SECOND+2
MOV  WORD PTR SUM+2,AX
```

解 3 采用直接寻址，双字运算只需 3 条指令。

```
MOV  EAX,FIRST
ADD  EAX,SECOND
MOV  SUM,EAX
```

(5) 二进制数加 1 指令(Increase)

格式：INC 目标操作数

功能：目标操作数+1→目标操作数。

(6) 二进制数减 1 指令(Decrease)

格式：DEC 目标操作数

功能：目标操作数-1→目标操作数。

(7) 二进制数求补指令(Neglect)

格式：NEG 目标操作数

功能：0-目标操作数→目标操作数。

说明：

① 以上 3 条指令，其目标操作数可以是 8 位、16 位或 32 位的寄存器操作数、内存操作数。

② 如果是用间址、基址、变址、基址加变址寻址的内存操作数，必须用 PTR 运算符明确说明其属性。例如：

INC	[BX]	;错误
INC	BYTE PTR [BX]	;对 DS: [BX]字节单元加 1
INC	WORD PTR [BX]	;对 DS: [BX]字单元加 1
INC	DWORD PTR [BX]	;对 DS: [BX]双字单元加 1

③ INC、DEC 指令执行后影响 A、O、P、S、Z 5 个标志,但对 C 标志没有影响。

④ NEG 指令执行后,影响 A、C、O、P、S、Z 6 个标志。

NEG 指令求出目标操作数的负数。如果原来的操作数为正数,执行 NEG 指令后,变成负数(用补码表示)。反之,原来的操作数为负数(用补码表示),则执行 NEG 指令后就变成正数。但有一特例,以单字节数为例:如果原来的操作数为 80H(即-128),执行 NEG 指令后,仍为 80H,但此时溢出标志为 1。另外,只有当操作数为 0 时,执行 NEG 指令后,C 标志才为 0,对于其他数,执行 NEG 指令后,C 标志总为 1。

(8) 交换加法指令

格式: XADD 目标操作数,源操作数

说明: 目标操作数是 8、16 或 32 位的寄存器操作数、内存操作数,源操作数只能是与目标等长的寄存器操作数。

功能: 首先进行源、目标操作数互换,然后完成源操作数+目标操作数→目标操作数。指令执行后的源操作数是指令执行前的目标操作数。

(9) 无符号二进制数乘法指令(Multiply)

格式: MUL 乘数

功能与说明:

① 乘数和被乘数必须是等长的无符号二进制数,乘积为双倍长。指令格式中写的是乘数,它可以是 8 位、16 位或 32 位的寄存器操作数、内存操作数。

② 字节相乘,被乘数默认在 AL 中,乘积存入 AX 寄存器,若高位积 AH 为 0,则 C 标志、O 标志置 0,否则置 1。

③ 字相乘,被乘数默认在 AX 中,乘积存入 DX、AX 寄存器,若高位积 DX 为 0,则 C 标志、O 标志置 0,否则置 1。

④ 双字相乘,被乘数默认在 EAX 中,乘积存入 EDX、EAX 寄存器,若高位积 EDX 为 0,则 C 标志、O 标志置 0,否则置 1。

(10) 有符号二进制数乘法指令(Integer Multiply)

IMUL 指令,默认其乘数被乘数均为有符号的补码数,最高一位表示符号,0 为正,1 为负,指令执行后,若 O 标志为 1,表示有溢出。

格式 1: IMUL 乘数

功能: 乘数、被乘数的预置、乘积的存放均与 MUL 指令相同。

格式 2: IMUL 目标操作数,源操作数

说明: 目标操作数可以是 16 位或 32 位的寄存器操作数、内存操作数,源操作数是与目标等长的立即数、寄存器操作数或内存操作数,但源、目标不能同为内存操作数。

功能: 源操作数×目标操作数→目标操作数。例如:

MOV BX, -6

IMUL BX,5 ;则 BX=FFE2H=-30

格式 3: IMUL 目标操作数,源操作数,立即数

功能: 源操作数 \times 立即数 \rightarrow 目标寄存器

说明: 目标操作数只能是 16 位或 32 位的通用寄存器,源操作数是与目标操作数等长的寄存器操作数或内存操作数,立即数可以为 16 位或 32 位,而且与源、目标等长,8 位立即数能自动进行符号扩展转换成 16 位或 32 位立即数。例如:

MOV EAX,32

IMUL EBX,EAX,4 ;则 EBX=00000080H,EAX=00000020H

(11) 无符号二进制数除法指令(Divide)

格式: DIV 除数

功能与说明:

① 指令格式中写出的是除数,被除数应当是除数的双倍字长。该指令根据除数的字长,可以执行字节除法、字除法和双字除法 3 种操作。

② 字节除法: 除数是 8 位的寄存器操作数或内存操作数,被除数默认在 AX 寄存器中。除法操作后,AL 中为商数,AH 中为余数。

③ 字除法: 除数为 16 位的寄存器操作数或内存操作数,被除数默认在 DX 和 AX 寄存器中(DX 为高 16 位)。除法操作后,AX 中为商数,DX 中为余数。

④ 双字除法: 除数为 32 位的寄存器操作数或内存操作数,被除数默认在 EDX 和 EAX 中(EDX 为高 32 位),除法操作后,EAX 中为商数,EDX 中为余数。

(12) 有符号二进制数除法指令(Integer Division)

格式: IDIV 除数

功能与说明:

同 DIV,但除数、被除数和商值都是有符号补码数,余数和被除数的属性相同,如果商值超出范围自动产生 0 型中断。

(13) 符号扩展指令

格式 1: CBW

功能: 将 AL 中的符号位扩展到 AH 中。

格式 2: CWD

功能: 将 AX 中的符号位扩展到 DX 中。

格式 3: CWDE

功能: 将 AX 中的符号位扩展到 EAX 的高 16 位

格式 4: CDQ

功能: 将 EAX 中的符号位扩展到 EDX 中

应用: 如果被除数不是除数的双倍字长,在进行有符号数除法时,需要用上述指令对被除数进行符号扩展,然后再进行有符号数除法。

(14) 比较指令(Compare)

格式: CMP 目标操作数,源操作数

功能：目标操作数-源操作数，但结果不回送目标，而是根据结果设置 A、C、O、P、S、Z 6 种状态标志。

说明：

① 目标操作数可以是 8 位、16 位或 32 位的寄存器操作数、内存操作数，源操作数是与目标操作数等长的立即数、寄存器操作数或内存操作数，但源、目标不可同为内存操作数。

② 如果目标是采用间址、基址、变址或基址加变址寻址的内存操作数，而源操作数是 8 位或 16 位立即数，则目标操作数必须用 PTR 运算符说明其属性。例如：

```
CMP    [BX], 'A'           ;错误
CMP    BYTE PTR [BX], 'A'   ;正确
```

应用：编程时用 CMP 指令比较两个数，然后根据指令产生的状态标志进行程序转移。

2. BCD 码调整指令

上述四则运算指令，其运算对象均为二进制数，CPU 按照“逢二进一，借一当二”的法则处理运算过程中的进位和借位。如果运算对象是十进制数（即 BCD 码数）怎么办？80x86 设置了一套“调整指令”。

请注意：调整指令并非用来进行运算。

为了理解调整指令的工作原理，表 3-4 列出了用二进制数加法指令完成十进制数运算的调整原则。

表 3-4 十进制数加法调整

笔 算	CPU 运算	加 法 调 整
$\begin{array}{r} 43 \\ + 55 \\ \hline 98 \end{array}$	$\begin{array}{r} 0100\ 0011 \\ \text{ADD) } 0101\ 0101 \\ \hline 1001, 1000 \end{array}$	C 标=0, A 标=0, 高低四位均没有出现非法 BCD 码, 结果正确, 不修正
$\begin{array}{r} 39 \\ + 49 \\ \hline 88 \end{array}$	$\begin{array}{r} 0011\ 1001 \\ \text{ADD) } 0100\ 1001 \\ \hline 1000, 0010 \\ +) \quad 0110 \\ \hline 1000, 1000 \end{array}$	低 4 位有进位, 即 A 标=1, 对运算结果加 06H 修正
$\begin{array}{r} 63 \\ + 54 \\ \hline 117 \end{array}$	$\begin{array}{r} 0110\ 0011 \\ \text{ADD) } 0101\ 0100 \\ \hline 1011, 0111 \\ +) \quad 0110 \\ \hline 1, 0001, 0111 \end{array}$	高 4 位出现非法 BCD 码数, 对运算结果加 60H 修正
$\begin{array}{r} 87 \\ + 86 \\ \hline 173 \end{array}$	$\begin{array}{r} 1000\ 0111 \\ \text{ADD) } 1000\ 0110 \\ \hline 1, 0000, 1101 \\ +) \quad 0110\ 0110 \\ \hline 1, 0111, 0011 \end{array}$	因为 C 标=1, 低 4 位出现非法 BCD 码, 对运算结果加 66H 修正

从表中列举的部分例题可以看出：用二进制加法指令完成十进制数运算，结果往往是错误的，道理很简单：

因为十进制数(BCD 码数)加法法则是“逢十进一”，即低四位大于 1001 时，就应当向 D_4 位“进一”。但是二进制数加法指令遵循“逢二进一”的法则，只有当低四位大于 16 时，才向 D_4 位“进一”，两种数制的进位值相差为 6，因此造成错误。

另一方面，在二进制数数列中出现 1010~1111 是有意义的，而在 BCD 码数列中出现 1010~1111 是非法 BCD 码。

综合以上分析，当使用二进制数运算指令，进行十进制数运算时，必须对结果进行有条件的修正，这种修正不需要程序员去做，80x86 专门设置了一套“调整指令”，调整指令会自动对运算结果进行修正。

(1) 组合十进制数加法调整指令(Decimal Adjust for Addition)

格式：DAA

功能：针对 AL 寄存器中的两个组合十进制数之和进行修正，得到正确的组合十进制数。

说明：

① 组合十进制数，又称压缩的十进制数，即两位 BCD 码数。

② CPU 执行 DAA 指令时，完成下列操作：

判断 DAA 指令执行前的 AL、A 标志和 C 标志。

若 AL 低 4 位大于 1001 或者 A 标志 = 1，则 $AL + 06H \rightarrow AL$ ， $1 \rightarrow A$ 标志；否则不修正，A 标志保持不变。

若 AL 高 4 位大于 1001 或者 C 标志 = 1，则 $AL + 60H \rightarrow AL$ ， $1 \rightarrow C$ 标志；否则不修正，C 标志保持不变。

③ 鉴于以上原因，在 DAA 指令之前执行的二进制数加法指令必须以 AL 为目标寄存器，而且必须能正确地影响 A 标志，C 标志。

也就是讲只能执行：

```
ADD    AL,源操作数
DAA
.....
```

或者执行：

```
ADC    AL,源操作数
DAA
.....
```

不能执行：

```
INC    AL
DAA
.....
```

④ DAA 指令执行后影响 A、C、P、S、Z 对 O 标志未定义。

【例 3.4.1】 假设数据段 BCD1 和 BCD2 字单元均为组合 BCD 码,计算这两个 BCD 码数之和,存入 SUM 字单元中。

数据段:

```
BCD1      DW      2345H
BCD2      DW      5678H
SUM       DW      ?
```

代码段:

```
.....
MOV      AL,BYTE PTR BCD1
ADD      AL,BYTE PTR BCD2      ;低 8 位相加
DAA      ;调整
MOV      BYTE PTR SUM,AL      ;存低 8 位之和
MOV      AL,BYTE PTR BCD1+1
ADC      AL,BYTE PTR BCD2+1    ;高 8 位相加
DAA      ;调整
MOV      BYTE PTR SUM+1,AL     ;存高 8 位之和
.....
```

(2) 组合十进制数减法调整指令(Decimal Adjust for Subtraction)

格式: DAS

功能: 针对 AL 中的两个组合十进制数之差进行修正,得到正确的用组合十进制数表示的差值。

说明:

① 80x86 允许用 SUB 或 SBB 指令直接进行两个组合十进制数的减法,但需要在减法指令后用 DAS 进行调整。

② DAS 要判断当前的 AL 寄存器和 A 标志、C 标志。

若 AL 低 4 位大于 1001,或 A 标志=1,则: $AL-06H \rightarrow AL$, $1 \rightarrow A$ 标志;否则不修正,A 标志不变。

若 AL 高 4 位大于 1001,或 C 标志=1,则: $AL-60H \rightarrow AL$, $1 \rightarrow C$ 标志;否则不修正,C 标志不变。

③ 针对 BCD 码数执行 SUB 或 SBB,若被减数大于或等于减数,用 DAS 调整后,C 标志=0,AL 寄存器中即为差值的组合十进制数。若被减数小于减数,结果应为负数,但是 BCD 码数无法表示负数,用 DAS 调整后 C 标志=1,AL 寄存器中得到的是差值相对于模 100 的“补数”。

例如两个 BCD 码相减: $56H-68H = -12H$,用 DAS 调整之后,C 标志=1,AL=88H,相对于模 100 而言 88H 是 $-12H$ 的“补数”。

④ DAS 指令执行后,影响 A、C、P、S、Z 标志,对 O 标志未定义。

(3) 未组合十进制数加法调整指令(Unpacked BCD Adjust for Addition)

格式: AAA

功能：针对 AL 寄存器中的两个未组合十进制数之和进行修正。从而在 AH 中得到十位 BCD 码，AL 中得到个位 BCD 码，AH、AL 均为未组合十进制数。

说明：

① 未组合十进制数，又称非压缩十进制数，即字节的高 4 位恒为 0000。低 4 位为 0000~1001 的 BCD 码数。

② CPU 执行 AAA 指令时，首先判断指令执行前的 AL、A 标志。若 AL 低 4 位大于 1001，或 A 标志 = 1，则：AL + 6 → AL，AL 高 4 位清零；AH + 1 → AH；1 → A 标志 → C 标志。

③ 执行 AAA 之前程序员应使 AH = 0，AAA 执行后：

AH = 十位和的未组合 BCD 码数。

AL = 个位和的未组合 BCD 码数。

影响 A 标志和 C 标志，但对 O、P、S、Z 4 个标志未定义。

(4) 未组合十进制数减法调整指令(Unpacked BCD Adjust for Subtraction)

格式：AAS

功能：AAS 指令紧跟在 SUB/SBB 指令之后，对 AL 中的两个未组合十进制数的差值进行修正。当被减数小于减数时，AAS 执行后 C 标志为 1，AH 中的内容减 1，AL 中的差值是相对于模 10 的“补数”。如果被减数大于或等于减数，AAS 执行后 C 标志为 0，AH 不变，对 AL 中的差值不做修正。例如：

```
MOV    AX,0004H
SUB     AL,05H           ;04H-05H→AL
AAS                      ;C标志=1,AH=FFH,AL=09H
```

(5) 未组合十进制数乘法调整指令(Unpacked BCD Adjust for Multiply)

格式：AAM

功能：对 AX 中的两个未组合十进制数之积进行调整，调整后 AH 中等于十位积的未组合十进制数，AL 中等于个位积的未组合十进制数。

说明：

① 80x86 允许用 MUL 指令实现两个未组合十进制数的乘法，然后再用 AAM 对乘积进行修正。

② 用 MUL 指令实现两个未组合十进制数相乘，其乘积在 AX 寄存器中，最大为 81，它是用二进制表示的。

AAM 指令对 AX 中的内容进行以下操作：

AL ÷ 10 的商数 → AH，余数 → AL。从而经过 AAM 调整之后，AH 中为乘积的十位数，AL 中为乘积的个位数，它们都是高 4 位恒为 0000 的未组合十进制数。

③ 指令执行后影响 P、S、Z 标志，对 A、C、O 标志未定义。

(6) 未组合十进制数除法调整指令(Unpacked BCD Adjust for Division)

格式：AAD

功能：CPU 执行：AH × 10 + AL → AL，0 → AH。从而把 AX 中两位未组合十进制

数(AH 中为十位数,AL 中为个位数)转换成等值的二进制数。

指令执行后影响 P、S、Z 标志,对 A、C、O 标志未定义。

应用: DIV 指令是二进制数除法指令。用 DIV 指令也可以进行十进制数字节除法。当被除数是两位未组合十进制数,除数是一位未组合十进制数的时候,必须先用 AAD 指令把被除数转换成等值的二进制数,然后再用 DIV 指令做除法,DIV 执行完毕,AL 中得到商值的未组合十进制数,AH 中为余数的未组合十进制数。

【例 3.4.2】 编程实现十进制数除法: $86 \div 3$ 。

程序段如下:

```
MOV     BL,03H
MOV     AX,0806H
AAD                      ;0806H 转换成等值的二进制数,AX=0056H
DIV     BL                ;AL 为二进制商数 11100,AH=余数
```

在该程序之后,首先保存 AH 中的余数,再执行一条 AAM 指令,就可将 AL 中的二进制商数转换成未组合 BCD 码(AX=0208H)。

【小结】

综上所述,进行十进制数加、减、乘运算的时候,先用二进制数运算指令进行运算(结果为二进制数),然后再用相关的调整指令,对结果进行修正,修正之后的结果就是用 BCD 码表示的十进制数。欲进行十进制数除法运算,先用除法调整指令把被除数转换成等值的二进制数,然后再用 DIV 指令做除法,DIV 执行完毕,AL 中得到商值的二进制数,AH 中为余数的二进制数。

3.4.3 转移和调用指令

80486 的转移指令种类繁多,按照转移条件,可分为无条件转移和有条件转移。按照转移范围可分为段内转移和段间转移。按照获取转移地址的方法,又分为直接转移和间接转移。调用指令的分类和转移指令类似。

1. 无条件转移指令

功能: 无条件转移,执行指定标号处的指令。

格式 1: 段内直接转移 JMP 标号
JMP SHORT 标号

说明:

① 指令格式中的标号是转移地址标号。

② SHORT 是汇编语言提供的运算符,仅用于 JMP 指令,表明是短距离转移,其转移范围相对于指令地址而言在 $+129 \sim -126$ 个字节之间,超越此范围,汇编时给出错误信息。不含 SHORT 的指令,其转移范围可以覆盖整个逻辑段。

格式 2: 段内间接转移 JMP 寄存器操作数
JMP 内存操作数

说明：程序员应预先把转移地址的偏移地址分量写入等长的寄存器或内存单元，而在指令格式中只须写出存放偏移地址的寄存器名称或该内存的寻址方式，CPU 执行指令时，将寄存器或内存单元的偏移地址写入 IP 或 EIP，从而实现转移，这就是“间接”转移的含义。

虽然 80486 允许对内存单元采用 16 位寻址或者 32 位寻址，但是在实地址模式下，一个逻辑段的体积最大为 64KB。因此在实地址模式下最好采用 16 位寻址转移。例如：

设数据段：

```
POINTER DW    P11           ;汇编后 P11 的 16 位偏移地址→POINTER 字单元；
          DW    P12
```

代码段：对 DS 初始化

```
MOV     BX,OFFSET POINTER
JMP     [BX]           ;DS: [BX] →IP,转移到 P11 程序段
.....
MOV     SI,POINTER+2
JMP     SI             ;SI→IP,转移到 P12 程序段
P11:    .....
P12:    .....
```

格式 3：段间直接转移 JMP 标号
 段间间接转移 JMP 内存操作数

说明：

① 在模块化程序中，从一个模块转移到另一个模块需要执行段间转移指令。段间直接转移是常用的格式，但此时对转移标号要做两项说明：即在本模块中用 EXTRN 伪指令说明转移标号是“外部变量名”，而在转移目标的模块中用 PUBLIC 伪指令说明转移标号是“公共变量”。

② 实地址模式下，段间间接转移指令的功能示范如下：

数据段：

```
ADDR      DD            12345678H      ;32 位转移地址
```

代码段：对 DS 初始化

```
          JMP            ADDR            ;1234H→CS,5678H→IP 实现段间转移
```

用户程序怎样使用段间间接转移指令？典型的用法是在用户的中断服务程序中，当中断服务结束，需要转移到某个系统中断服务程序时，根据被转换的中断向量进行段间转移。

2. 条件转移指令

条件转移最常见的用法是紧跟在比较指令之后，测试比较指令产生的状态标志，当条件满足时执行指定标号处的指令，否则顺序执行后继指令。

8086/8088 的条件转移指令,其转移范围很小,相对于指令地址而言,其转移范围仅在+129~-126 之间,从 80386 开始扩大了转移范围,实地址模式下能够转移到代码段的任何位置。

条件转移指令有统一的格式,即:

操作码助记符 转移地址标号

各种转移条件隐含在操作码助记符当中,有的指令又有几种等价的操作码助记符,下面分类介绍操作码助记符及其功能。

(1) 按标志位的当前状态转移

操作码助记符	指令功能	等价助记符
JC	当前 C 标志为 1 转移	JB/JNAE
JNC	当前 C 标志为 0 转移	JNB/JAE
JZ	当前 Z 标志为 1 转移	JE
JNZ	当前 Z 标志为 0 转移	JNE
JS	当前 S 标志为 1 转移	
JNS	当前 S 标志为 0 转移	
JP	当前 P 标志为 1 转移	JPE
JNP	当前 P 标志为 0 转移	JPO
JO	当前 O 标志为 1 转移	
JNO	当前 O 标志为 0 转移	

(2) 无符号数条件转移

在执行了两个无符号数的比较指令之后,用下列格式完成条件转移:

操作码助记符	指令功能	等价助记符
JA	被减数大于减数转移	JNBE
JNA	被减数小于或等于减数转移	JBE
JNC	被减数大于或等于减数转移	JNB/JAE
JC	被减数小于减数转移	JB/JNAE

(3) 有符号数条件转移

在执行了两个有符号数的比较指令之后,用下列格式完成条件转移:

操作码助记符	指令功能	等价助记符
JG	被减数(真值)大于减数(真值)转移	JNLE
JGE	被减数(真值)大于或等于减数(真值)转移	JNL
JL	被减数(真值)小于减数(真值)转移	JNGE
JLE	被减数(真值)小于或等于减数(真值)转移	JNG

(4) 循环控制转移

操作码助记符	指令功能	等价助记符
LOOP	(E)CX-1→(E)CX,若(E)CX 不为 0 转移	
LOOPZ	(E)CX-1→(E)CX,若(E)CX 不为 0 且 Z 标志=1 转移	LOOPE
LOOPNZ	(E)CX-1→(E)CX,若(E)CX 不为 0 且 Z 标志=0 转移	LOOPNE
JCXZ	测试 CX, 若 CX 为 0 转移	
JECXZ	测试 ECX, 若 ECX 为 0 转移	

80486 循环控制指令,还可以使用 ECX 做为隐含的循环计数器而操作码助记符不变,即便是 80486,循环控制指令也只能做短距离转移,即相对于指令地址而言,转移范围为+129~-126 字节。

备注:

JB	~ Jump on Below	低于转移
JNAE	~ Jump on Not Above or Equal	不高于或不等于转移
JNB	~ Jump on Not Below	不低于转移
JAЕ	~ Jump on Above or Equal	高于或等于转移
JE	~ Jump on Equal	等于转移
JNE	~ Jump on Not Equal	不等于转移
JPE	~ Jump on Parity Even	偶转移
JPO	~ Jump on Parity Odd	奇转移
JA	~ Jump on Above	高于转移
JNBE	~ Jump on Not Below or Equal	不低于或不等于转移
JNA	~ Jump on Not Above	不高于转移
JBE	~ Jump on Below or Equal	低于或等于转移
JNLE	~ Jump on Not Less or Equal	不小于或不等于转移
JNL	~ Jump on Not Less	不小于转移
JNGE	~ Jump on Not Greater or Equal	不大于或不等于转移
JNG	~ Jump on Not Greater	不大于转移

3. 子程序调用与返回

在一个源程序中,如果多次用到某一项操作(例如把多个二进制数转换成十进制数显示),为了简化程序,通常设计一个子程序供调用。

子程序是相对独立的程序,调用它的程序称为主程序,或调用程序,子程序允许有多个入口,多个出口。子程序执行完毕必须返回调用程序的断点,执行调用指令的后继指令。

子程序又称为过程,按照汇编语言的要求,通常在源程序中用“过程定义语句”定义一个过程,过程名就是子程序名。如果子程序和调用它的主程序同属一个代码段则该子程序具有 NEAR(近)属性,否则子程序应设置 FAR(远)属性。有关过程定义语句的详情请参阅第 4 章。

子程序调用指令分为段内直接调用、段内间接调用、段间直接调用和段间间接调用 4 种格式,子程序返回指令有带参数和不带参数两种格式。

(1) 段内直接调用指令

格式: CALL 过程名

指令经过汇编之后,过程名即为子程序第一条指令的偏移地址(即入口地址)。

(2) 段内间接调用指令

格式: CALL 寄存器操作数

CALL 内存操作数

例如: CALL BX ;程序员应预先把子程序入口的偏移地址→BX
CALL POINTER ;应预先把子程序入口的偏移地址→POINTER
;字单元

CPU 执行段内调用指令时,首先把断口偏移地址压入堆栈,为子程序返回做准备,然后把子程序入口的偏移地址→IP,从而转入子程序。断口地址即是 CALL 指令的后继指令地址,对于段内调用指令,压入堆栈的仅仅是断口地址的偏移地址分量。

(3) 段间直接调用指令

格式: CALL 过程名

(4) 段间间接调用指令

格式: CALL 内存操作数

CPU 执行段间调用指令时,也是先把断口地址压入堆栈(但此时的断口地址包含段基址和偏移地址两个分量,是“全地址”压栈),然后再把子程序入口的偏移地址→IP,入口的段基址→CS,从而转向另一个代码段的子程序。有两种情况,需要使用段间调用指令:

① 在一个源程序中,从一个代码段调用另一个代码段中的子程序(一个源程序设置两个代码段,这种编程风格是不多见的),若采用段间直接调用,调用指令必须用 PTR 说明被调用的子程序是远过程,否则汇编时,给出错误信息。与此同时被调用的子程序必须写明它具有 FAR 属性,否则子程序执行完毕就不能返回断点,下面给出这种编程风格的实例,请注意段间直接调用和段间间接调用的使用方法,该程序执行后,屏幕的当前位置显示两个 A。

```
.486
DATA    SEGMENT USE16
POINTER DD      DISP
DATA    ENDS
CODE1   SEGMENT USE16
        ASSUME  CS: CODE1,DS: DATA
BEG:    MOV     AX,DATA
        MOV     DS,AX
        CALL    FAR PTR DISP      ;段间直接调用
        CALL    POINTER           ;段间间接调用
        MOV     AH,4CH
        INT     21H
CODE1   ENDS
CODE2   SEGMENT USE16
        ASSUME  CS: CODE2
DISP    PROC     FAR               ;定义 DISP 子程序具有 FAR 属性
        MOV     AH,2
        MOV     DL,'A'
        INT     21H
        RET
```

DISP ENDP
CODE2 ENDS
 END BEG

② 在模块化程序中,从一个模块调用另一个模块中的子程序(编写大型程序,采用模块化程序设计是很好的编程风格),也必须使用段间调用指令,此时在调用模块中必须用 EXTRN 伪指令说明所调用的子程序,其过程名是外部变量,并且具有“FAR”属性,而在被调用模块中,必须用 PUBLIC 伪指令说明被调用的子程序,其过程名是“公共变量”。当然,被调用的子程序必须在过程定义语句中写明具有“FAR”属性,否则子程序执行完毕不能返回断点。

(5) 无参数返回指令

格式: RET

(6) 有参数返回指令

格式: RET N(N 为偶数)

功能与说明: 返回指令是子程序的出口指令,在具有 NEAR 属性的子程序中,RET 指令从栈顶弹出 2 个字节→IP,然后 $SP+2 \rightarrow SP$ 。在具有 FAR 属性的子程序中,RET 指令从栈顶弹出 4 个字节→IP、CS,然后 $SP+4 \rightarrow SP$ 。如果在执行 RET 指令之前,栈顶元素仍然是(这一点非常重要!)调用程序的断口地址,则 RET 指令执行后,能够返回调用程序断点,否则不能!

近过程和远过程中的 RET 指令,格式相同,但汇编后生成的目标代码不一样,前者为 C3H,后者为 CBH。

带参数的返回指令,首先完成 RET 指令的操作,然后把堆栈指针向高地址方向再额外调整 N 个字节,即再执行 $SP+N \rightarrow SP$ 。在什么情况下使用带参数的返回指令? 如果调用程序使用堆栈向子程序传递数据,那么子程序返回时,应使用“RET N”指令,在返回调用程序的时候,也顺便“清除”堆栈。本书第 5 章例 5.7.1 解法 2 有应用示范。

4. 中断调用与返回

(1) 中断调用指令

格式: INT n (n=0~255)

n 称为中断类型码,CPU 执行该指令首先将标志寄存器的内容以及“INT n”指令的断口地址(INT n 指令所在代码段的段基址和断口——INT n 后继指令——的偏移地址)共 6 个元素压入栈顶,然后转入系统 n 型中断服务程序。该指令又称软件中断指令。

(2) 中断返回指令

格式: IRET

该指令为中断服务程序的返回指令。CPU 执行该指令时,依次从栈顶弹出 6 个元素→IP、CS、F,从而返回调用程序断点。

由于中断调用和中断返回指令涉及中断系统的许多概念,在本节中只能简要介绍它们的功能,第 9 章再做详细介绍。

3.4.4 逻辑运算和移位指令

逻辑运算是按位运算,这类指令(除 NOT 外)执行后,都要影响某些标志位。

1. 逻辑运算指令

(1) 取反指令

格式: NOT 目标操作数

功能: 目标操作数按位取反,不影响标志。

(2) 与指令

格式: AND 目标操作数,源操作数

功能: 源、目标操作数按位相“与”,结果→目标操作数,影响 P、S、Z 标志,A 标志未定义,C 标志、O 标志置 0。

(3) 或指令

格式: OR 目标操作数,源操作数

功能: 源、目标操作数按位相“或”,结果→目标操作数,影响 P、S、Z 标志,A 标志未定义,C 标志、O 标志置 0。

(4) 异或指令

格式: XOR 目标操作数,源操作数

功能: 源、目标操作数按位“异或”,结果→目标操作数,影响 P、S、Z 标志,A 标志未定义,C 标志、O 标志置 0。

说明:

① 上述指令的目标操作数可以是 8 位、16 位或 32 位的寄存器操作数、内存操作数,源操作数是与目标等长的立即数、寄存器操作数或内存操作数,但不允许源、目标同为内存操作数。

② 如果目标为间址、基址、变址、基址加变址寻址的内存操作数,而源为单字节或双字节立即数,则目标必须用 PTR 运算符说明是字节型或字型。

2. 移位指令

(1) 开环移位指令

算术左移: SAL 操作数,移位次数

算术右移: SAR 操作数,移位次数

逻辑左移: SHL 操作数,移位次数

逻辑右移: SHR 操作数,移位次数

(2) 闭环移位指令

含进位的循环左移: RCL 操作数,移位次数

含进位的循环右移: RCR 操作数,移位次数

不含进位的循环左移: ROL 操作数,移位次数

不含进位的循环右移: ROR 操作数,移位次数

功能与说明：

① 以上 8 条移位指令,其运算对象均为 8 位、16 位或 32 位的寄存器操作数、内存操作数,移位次数可以是立即数或者是预先存放在 CL 寄存器中的移位次数。

② 如果操作数是采用间址、基址、变址、基址加变址寻址的内存操作数,必须用 PTR 运算符说明它的属性是字节型、字型,还是双字型。

③ 图 3-3 形象地画出了各条指令的移位功能,其中算术左移(SAL)和逻辑左移(SHL),助记符虽然不同,但 CPU 执行的操作是一样的。下面举几例以做示范：

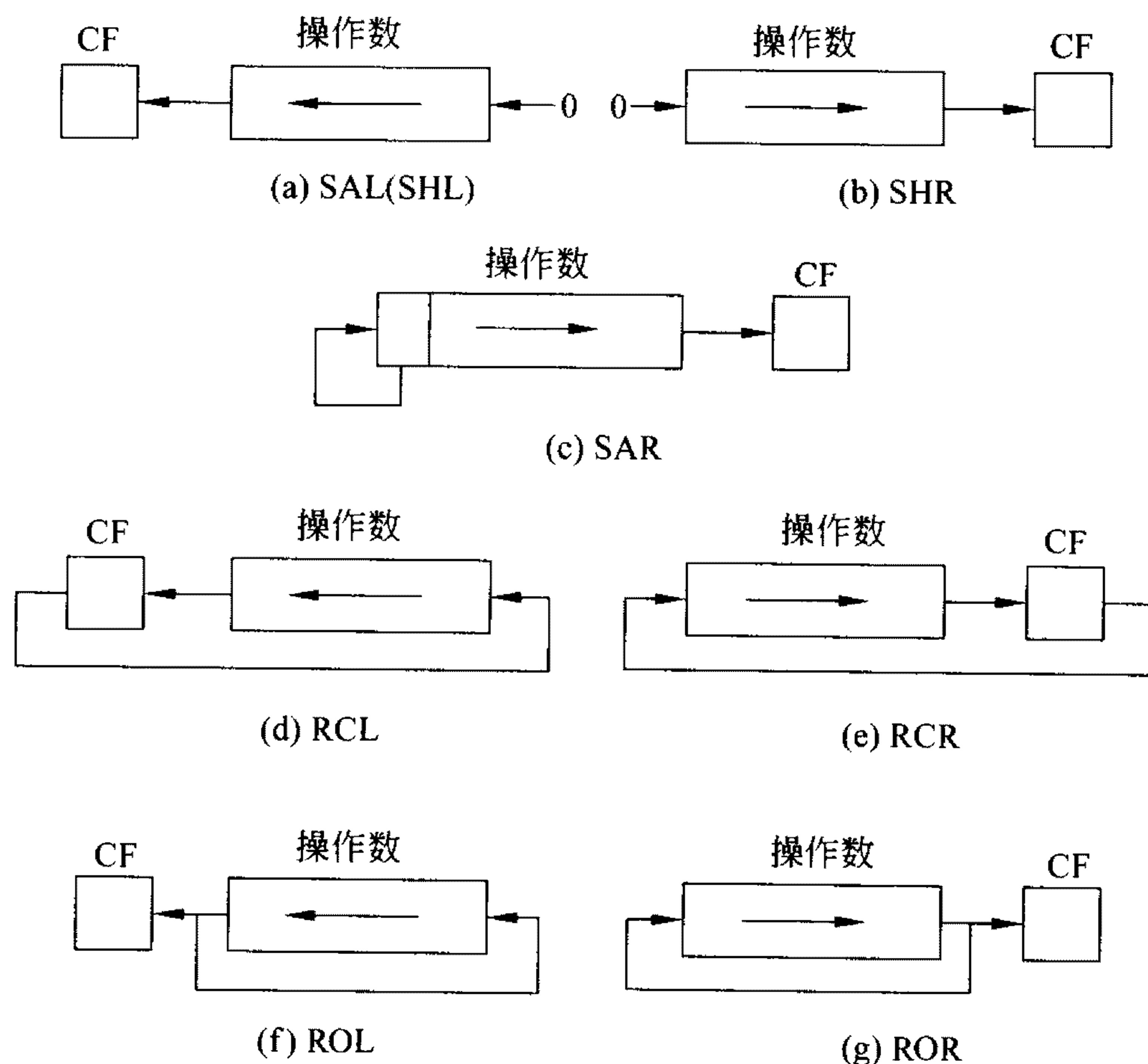


图 3-3 移位指令示意图

SAL AL,1;把 AL 内容左移一位, D_7 位移进 C 标志,(原来的 C 标志被覆盖),末位补 0,该指令将 AL 中的无符号数乘以 2,若 C 标志为 1,表示有溢出。

SAR WORD PTR BUF,1;这条指令把 BUF 字单元中的数右移一位,高位补进和原 D_{15} 位相同的值, D_0 位移进 C 标志,该指令将 BUF 单元中的双字节有符号数除以 2,并取其整数。

连续执行 MOV CL,16 和 ROL EAX,CL,与单独执行 ROL EAX,16 的结果是相同的,都是完成 EAX 高低 16 位的互换。

3. 测试与位测试指令

(1) 测试指令

格式：TEST 目标操作数,源操作数

功能：源、目标操作数相“与”,但结果不回送目标操作数,对标志位的影响和 AND 指令相同。

说明：使用时的注意事项和上述逻辑运算指令相同，该指令通常后跟转移指令，用来测试目标操作数的某一位或某些位。例如：

```

TEST    AL,20H
JNZ     NEXT1           ;AL 寄存器 D5 位=1 转移
.....
NEXT1:   .....
TEST    BUF,00011110B
JZ      NEXT2           ;BUF 单元的字节数 D4~D1 位都为 0 转移
.....
NEXT2:   .....
```

(2) 位测试指令

位测试	BT	目标操作数,源操作数
位测试置 0	BTR	目标操作数,源操作数
位测试置 1	BTS	目标操作数,源操作数
位测试取反	BTC	目标操作数,源操作数

功能与说明：

① 这是 80386 的新增指令，其目标操作数是 16 位或 32 位的寄存器操作数、内存操作数，源操作数是立即数或者是与目标等长的寄存器操作数。

② 这 4 条指令相同的功能是：测试目标操作数中的由源操作数指定的那一位，并将测试位的值送 C 标志。如果源操作数大于等于目标操作数字长，则源操作数除以目标操作数字长之后，其余数才是测试位。指令执行后，源操作数不变。

③ 这 4 条指令不同之处在于：BT 执行后，目标操作数不变，而 BTR、BTS、BTC 执行后，测试位分别被置 0、置 1、取反，例如：

```

BT      AX,0           ;AX 的 D0 位→C 标志
BT      AX,16          ;AX 的 D0 位→C 标志
BTR     EAX,31          ;EAX 的 D31 位→C 标志,然后 EAX 的 D31 位置 0
MOV     EAX,48
BTS     EBX,EAX         ;EBX 的 D16 位→C 标志,然后 1→EBX 的 D16 位,EAX 不变
```

4. 位扫描指令

位扫描指令有两种格式。

(1) 向前位扫描指令(bit scan forward)

格式：BSF 目标寄存器,源操作数

功能：从源操作数的最低位开始向高位搜索，将遇到的第一个“1”所在的位序号存入目标寄存器中。

(2) 向后位扫描指令(bit scan reverse)

格式：BSR 目标寄存器,源操作数

功能：从源操作数的最高位开始向低位搜索，将遇到的第一个“1”所在的位序号存入

目标寄存器中。

说明：

① 这两条也是 80386 的新增指令，其源操作数只能是 16 位、32 位的寄存器操作数或者内存操作数，目标寄存器必须与源操作数等长。指令执行后，源操作数不变。

例如：

```
MOV    EAX,12345678H
BSF    EBX,EAX           ;EBX=3,EAX 不变,Z 标志置 0
BSR    BX,AX             ;BX=14,AX 不变,Z 标志置 0
```

② 实验证明：如果源操作数为 0，则 Z 标志置 1，目标寄存器内容不变。如果源操作数为非 0，则 Z 标志置 0。

3.4.5 串操作指令

80x86 有 6 条串操作指令，它们是串传送、串比较、串搜索、串装入、串存储和 I/O 串操作，本小节仅介绍前 5 条。从 80386 开始各条串操作指令的功能均有所扩展，在原先的字节操作、字操作基础上，增加了双字操作，而且可以采用 16 位寻址和 32 位寻址。各种串操作指令虽然功能不同，但有许多共同之处，例如：每当对串中的一个“元素”操作之后，都要自动修改串地址指针，使其指向下一个元素。什么是元素？我们做这样的约定：在字节串中，一个字节就是一个元素，在字串中，2 个字节为一个元素，在双字串中，4 个字节为一个元素。

源串和目标串的存储及寻址方式都有隐含的规定，即：源串要放在数据段，目标串要放在 ES 附加段，在 16 位寻址操作下，CPU 自动用 SI 间址访问数据段，用 DI 间址访问 ES 附加段、用 CX 做为串计数器。在 32 位寻址操作下，CPU 自动用 ESI 间址访问数据段，用 EDI 间址访问 ES 附加段，用 ECX 做串计数器。

由于实地址模式下逻辑段的最大体积为 64KB，没有必要使用 32 位寻址，为了描述方便，在介绍指令功能的时候，均以 16 位寻址为基础。

1. 串传送指令 (Move String)

该指令把数据段的若干数据→附加段。CPU 对数据段自动使用 SI 间址取数，对 ES 附加段自动使用 DI 间址存数。

【基本型格式】

```
字节传送    MOVSB
字传送      MOVSW
双字传送    MOVSD
```

CPU 执行 MOVSB 时，把 DS: [SI] 的 1 个字节→ES: [DI] 的字节单元。

CPU 执行 MOVSW 时，把 DS: [SI] 的 2 个字节→ES: [DI] 的 2 个单元。

CPU 执行 MOVSD 时，把 DS: [SI] 的 4 个字节→ES: [DI] 的 4 个单元。

完成上述操作后，再根据方向标志(D 标志)自动修改 SI、DI(请参见表 3-5)。

表 3-5 修改串指针的操作

	字节操作		字操作		双字操作	
D 标志=0 为增址型	SI+1→SI	DI+1→DI	SI+2→SI	DI+2→DI	SI+4→SI	DI+4→DI
D 标志=1 为减址型	SI-1→SI	DI-1→DI	SI-2→SI	DI-2→DI	SI-4→SI	DI-4→DI

执行该指令之前应做如下准备工作：

欲进行增址型传送：

- ① 用 CLD 指令,使方向标志 $D=0$ 。
- ② 源串首址→DS: SI,目标区首址→ES: DI。

欲进行减址型传送：

- ① 用 STD 指令,使方向标志 $D=1$ 。
- ② 源串末址→DS: SI,目标区末址→ES: DI。

【有重复前缀的格式】

字节传送 REP MOVSB

字传送 REP MOVSW

双字传送 REP MOVSD

功能与说明：REP(Repeat 重复)仅仅是“前缀助记符”，不能单独使用。CPU 在执行有重复前缀的串传送指令时，除了完成基本型传送指令的操作之外，每传送一个元素都要自动完成 $CX-1 \rightarrow CX$ ，如果 CX 不为 0，传送后继元素，直到 $CX-1=0$ 时为止。

因此，和基本型指令相比，执行该指令前还应增加一项准备工作，即：执行字节串传送指令之前，应把要传送的字节数→CX。执行字串传送指令前，应把要传送的字数→CX。执行双字串传送前，应把要传送的双字的个数→CX。指令功能如图 3-4 所示。

2. 串装入指令 (load from String)

该指令把源串中的一个字节、一个字或一个双字装入 AL、AX 或 EAX，源串应放在数据段，CPU 自动用 SI 间址取数。

【基本型格式】

字节装入 LODSB

字装入 LODSW

双字装入 LODSD

说明：该指令把 DS: [SI] 中的一个字节→AL，一个字→AX 或一个双字→EAX 然后根据 D 标志自动修改 SI，该指令没有带前缀的格式。

3. 串存储指令 (Store in to String)

该指令把一个字节、一个字或一个双字写入目标区的若干单元。目标区应设置在

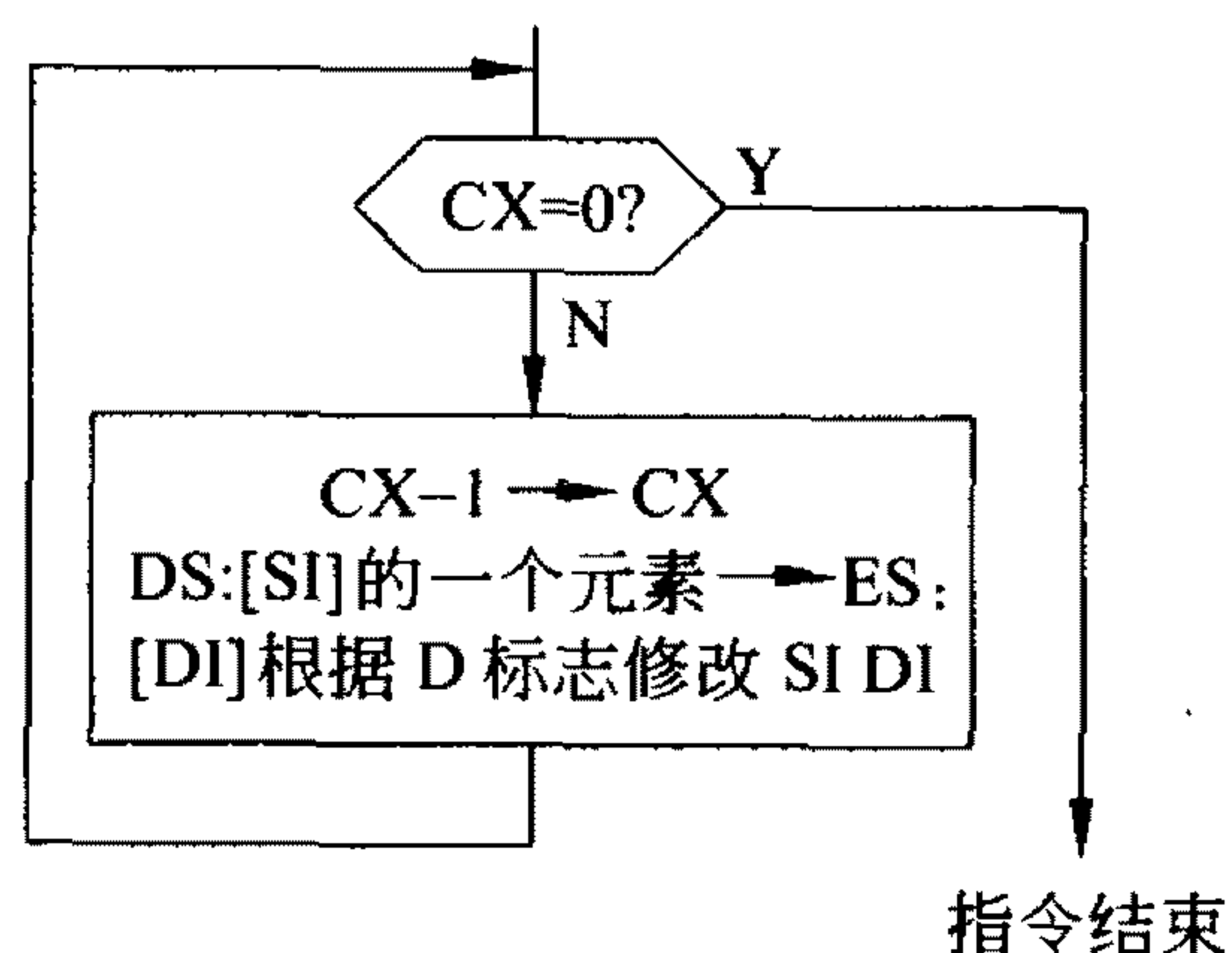


图 3-4 REP MOVSB
REP MOVSW
REP MOVSD 指令功能示意图

ES 附加段,CPU 自动用 DI 间址存数。

【基本型格式】

- 字节存储 STOSB
- 字存储 STOSW
- 双字存储 STOSD

CPU 执行 STOSB 时,把 AL→ES: [DI]的一个单元,执行 STOSW 时,把 AX→ES: [DI]的 2 个单元,执行 STOSD 时,把 EAX→ES: [DI]的 4 个单元,然后根据 D 标志自动修改 DI。

欲进行增址型存储应做如下准备工作:

- ① 用 CLD 指令使 D 标志置 0。
- ② 源操作数→AL、AX 或 EAX,目标区首址→ES: DI。

欲进行减址型存储应做如下准备工作:

- ① 用 STD 指令使 D 标志置 1。
- ② 源操作数→AL、AX 或 EAX,目标区末址→ES: DI。

【有重复前缀的格式】

- 字节存储 REP STOSB
- 字存储 REP STOSW
- 双字存储 REP STOSD

说明:

- ① 和基本型格式相比,指令执行前增加一项准备工作: 执行有重复前缀的串存储指令之前,应把欲存储的元素个数(即字节数、字数或双字个数)→CX。
- ② CPU 执行该指令时,除了完成基本型存储指令的操作之外,每存储一个元素,都要自动完成 CX-1→CX。如果 CX-1 不为 0,则存储下一个元素,CX-1=0 指令结束。

【例 3.4.3】 数据块传送。

将数据段 BLOCK 单元开始的 100 个字节依次传送到附加段 BUF 开始的内存区。

设数据段: BLOCK DB ××, ..., ××;100 个字节型数据

设附加段: BUF DB 100 DUP(?)

下面用 5 种方法编程,示范串指令的应用。

【用 MOV 指令编程】

对 DS、ES 初始化

```

MOV     SI,OFFSET BLOCK
MOV     DI,OFFSET BUF
MOV     CX,100
LAST:   MOV     AL,[SI]
        MOV     ES:[DI],AL
        INC     SI
        INC     DI
        LOOP    LAST
        .....

```

**【用 MOVSB 指令编程】**

对 DS、ES 初始化

```
MOV     SI,OFFSET BLOCK
MOV     DI,OFFSET BUF
MOV     CX,100
CLD
LAST:   MOVSB
        LOOP    LAST
        .....
```

【用 REP MOVSB 指令编程】

对 DS、ES 初始化

```
MOV     SI,OFFSET BLOCK
MOV     DI,OFFSET BUF
MOV     CX,100
CLD
REP     MOVSB
        .....
```

【用 LODSB,STOSB 指令编程】

对 DS、ES 初始化

```
MOV     SI,OFFSET BLOCK
MOV     DI,OFFSET BUF
MOV     CX,100
CLD
LAST:   LODSB
        STOSB
        LOOP    LAST
        .....
```

【用 REP MOVSD 指令编程】

对 DS、ES 初始化

```
MOV     SI,OFFSET BLOCK
MOV     DI,OFFSET BUF
MOV     CX,25
CLD
REP     MOVSD
        .....
```

4. 串比较指令 (Compare String)

该指令比较两串数据,源串必须在数据段,CPU 自动用 SI 间址访问,目标串必须在

ES 附加段, CPU 自动用 DI 间址访问。

【基本型格式】

字节串比较 CMPSB
 字串比较 CMPSW
 双字串比较 CMPSD

该指令的功能示意请见图 3-5(a)。

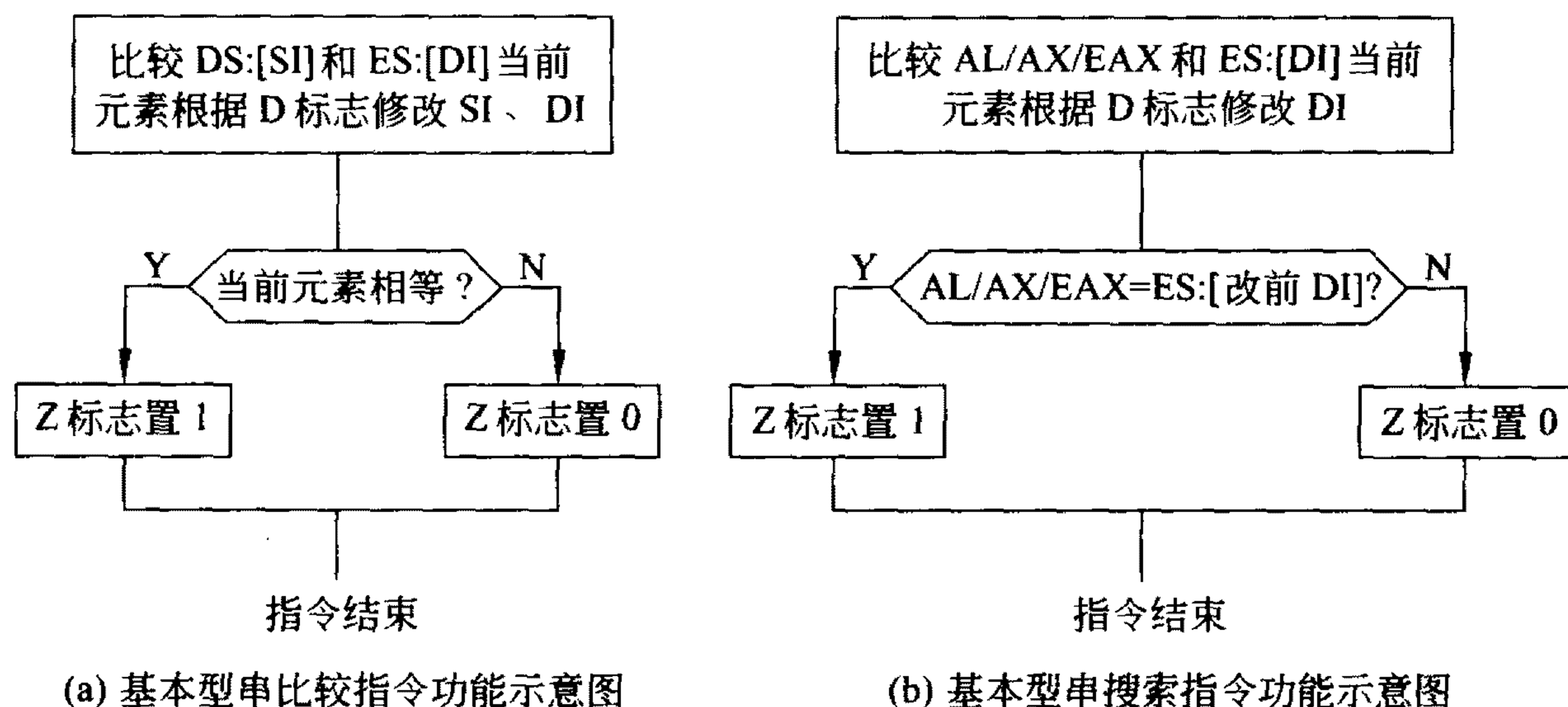


图 3-5 基本型串比较、串搜索指令功能示意图

指令执行后, SI、DI 都自动指向下一个元素, Z 标志为 1 表示当前元素相等, Z 标志为 0 表示当前元素不等。

该指令执行前应做的准备工作:

欲进行增址型比较:

用 CLD 指令使方向标志 D=0。源串首地址→DS: SI, 目标串首地址→ES: DI。

欲进行减址型比较:

用 STD 指令使方向标志 D=1。源串末地址→DS: SI, 目标串末地址→ES: DI。

【有重复前缀的格式 1】

字节串比较 REPE CMPSB (或 REPZ CMPSB)
 字串比较 REPE CMPSW (或 REPZ CMPSW)
 双字串比较 REPE CMPSD (或 REPZ CMPSD)

说明:

前缀助记符 REPE 和 REPZ 等价。

和基本型格式相比, 指令执行前的准备工作还应增加一项: 即把串中元素的个数→CX。

该指令的功能如图 3-6(a)所示。

当预置的串长度为 0 时(这是毫无意义的), 不进行任何操作而结束指令, 此时 Z 标志为指令执行前的状态。在预置的串长度不为 0 的前提下, CPU 比较 DS: [SI] 和 ES: [DI] 的当前元素, 并修改 SI、DI, 如果当前元素相等, 则比较下一个元素……指令结束后如果 Z 标志为 1, 表明两串数据相等, Z 标志为 0, 表明至少有一元素不相等。根据 D 标

志和 SI、DI 的值可以推断出是第几个元素不相等。

【有重复前缀的格式 2】

字节串比较 REPNE CMPSB (或 REPNZ CMPSB)

字串比较 REPNE CMPSW (或 REPNZ CMPSW)

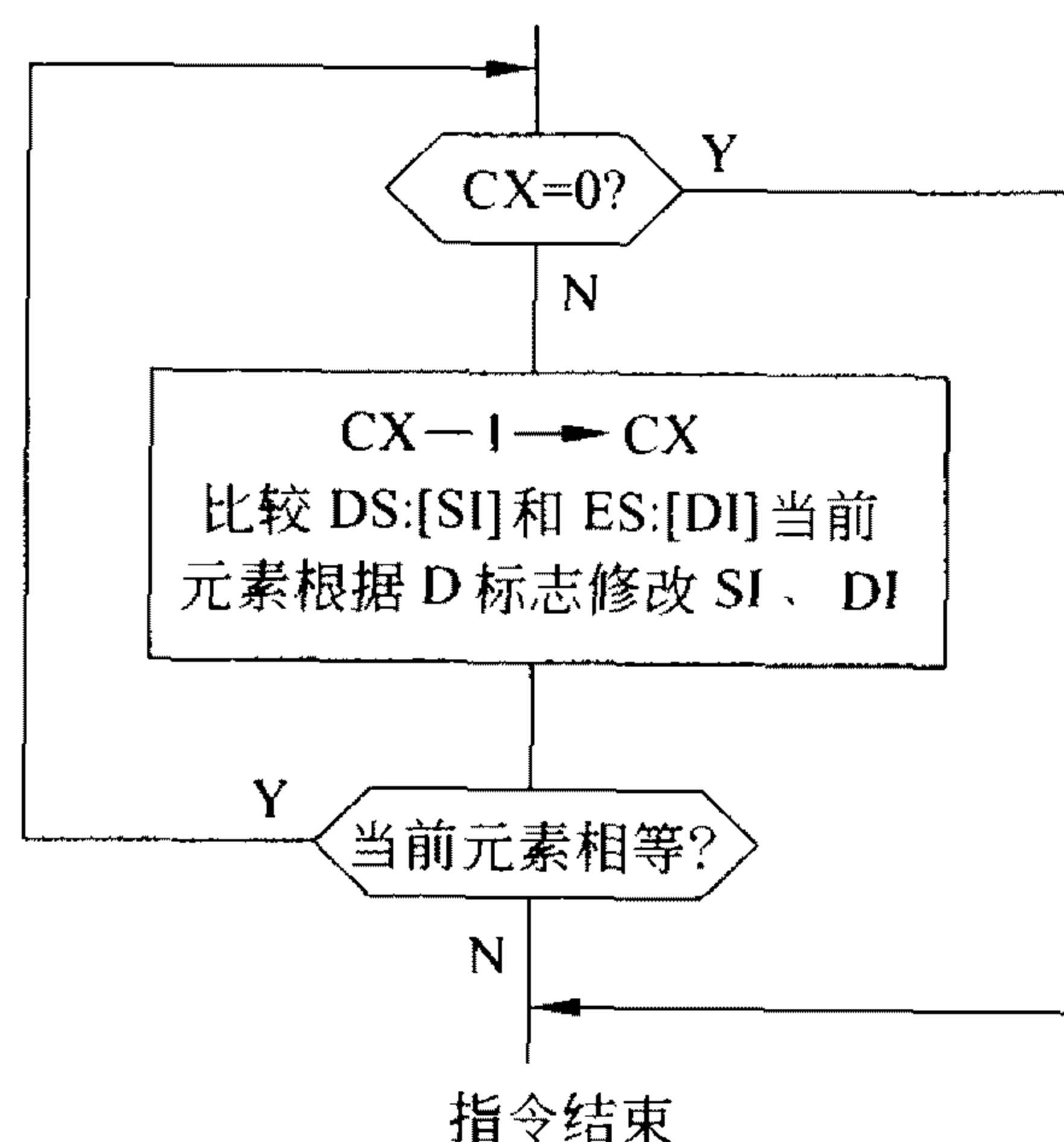
双字串比较 REPNE CMPDS (或 REPNZ CMPDS)

说明:

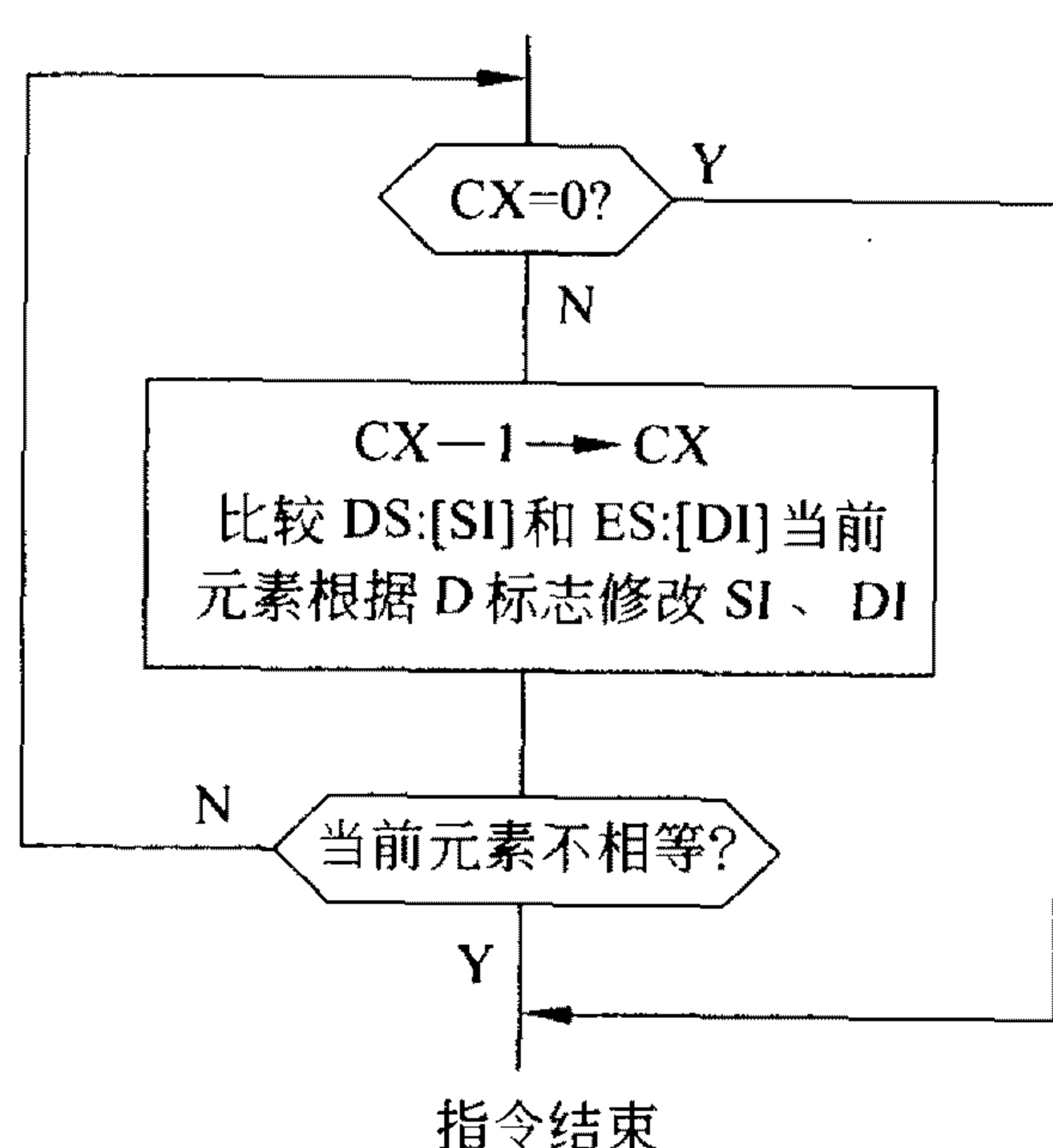
前缀助记符 REPNE 和 REPNZ 等价。

和基本型格式相比,指令执行前的准备工作应增加一项:即把串元素的个数→CX。

该指令的功能如图 3-6(b)所示,在预置的串长度不为 0 的前提下,指令结束后,若 Z 标志为 0,表示两串数据中对应的元素都不相同;若 Z 标志为 1,表示至少有一个元素是相同的。



REPE CMPSB
(a) REPE CMPSW 指令功能示意图
REPE CMPDS



REPNE CMPSB
(b) REPNE CMPSW 指令功能示意图
REPNE CMPDS

图 3-6 有重复前缀的串比较指令功能示意图

【例 3.4.4】字符串比较。

设数据区有两个字节串,串 1 的长度为 5,串 2 的长度为 10。

请判断:若串 2 的最后 5 个字符和串 1 相同,则置 FLAG 单元为‘Y’,否则置 FLAG 单元为‘N’。

设数据段	STRING1	DB	××, ..., ×× ; 串 1 长度=5
附加段	STRING2	DB	××, ..., ×× ; 串 2 长度=10
	FLAG	DB	'Y'
代码段	对 DS、ES 初始化		
	MOV	SI, OFFSET STRING1+4	
	MOV	DI, OFFSET STRING2+9	
	MOV	CX, 5	
	STD		

```

                REPE    CMPSB
                JZ      EXIT
                MOV     FLAG, 'N'
EXIT:          返回 DOS

```

5. 串搜索指令(Scan String)

该指令在目标串中搜索是否有指定的元素——该元素称为“关键字”。该指令要求目标串应放在 ES 附加段, CPU 自动用 DI 间址访问, 关键字应放在 AL、AX 或 EAX 寄存器中。

【基本型格式】

```

字节串搜索    SCASB
字串搜索      SCASW
双字串搜索    SCASD

```

该指令的功能如图 3-5(b)所示。指令执行后 DI 自动指向下一个元素, 若 Z 标志为 1 表明 ES: [改前 DI] 中的元素是关键字, 否则不是。指令执行前应做的准备工作:

欲进行增址型比较:

- ① 用 CLD 指令使方向标志 D=0。
- ② 要搜索的关键字放入 AL、AX 或 EAX 中。
- ③ 目标串首址→ES: DI。

欲进行减址型比较:

- ① 用 STD 指令使方向标志 D=1。
- ② 要搜索的关键字放入 AL、AX 或 EAX 中。
- ③ 目标串末址→ES: DI。

【有重复前缀的格式 1】

```

字节串搜索    REPE    SCASB      (或 REPZ SCASB)
字串搜索      REPE    SCASW      (或 REPZ SCASW)
双字串搜索    REPE    SCASD      (或 REPZ SCASD)

```

说明: 和基本型格式相比, 指令执行前需增加一项准备工作: 即把目标串中元素的个数→CX。该指令的功能如图 3-7(a)所示。

当预置的串长度不为 0 时, 指令执行后若 Z 标志=1, 表明目标串中每一个元素都是关键字, 若 Z 标志为 0, 表示目标串中至少有一个元素不是关键字。

【有重复前缀的格式 2】

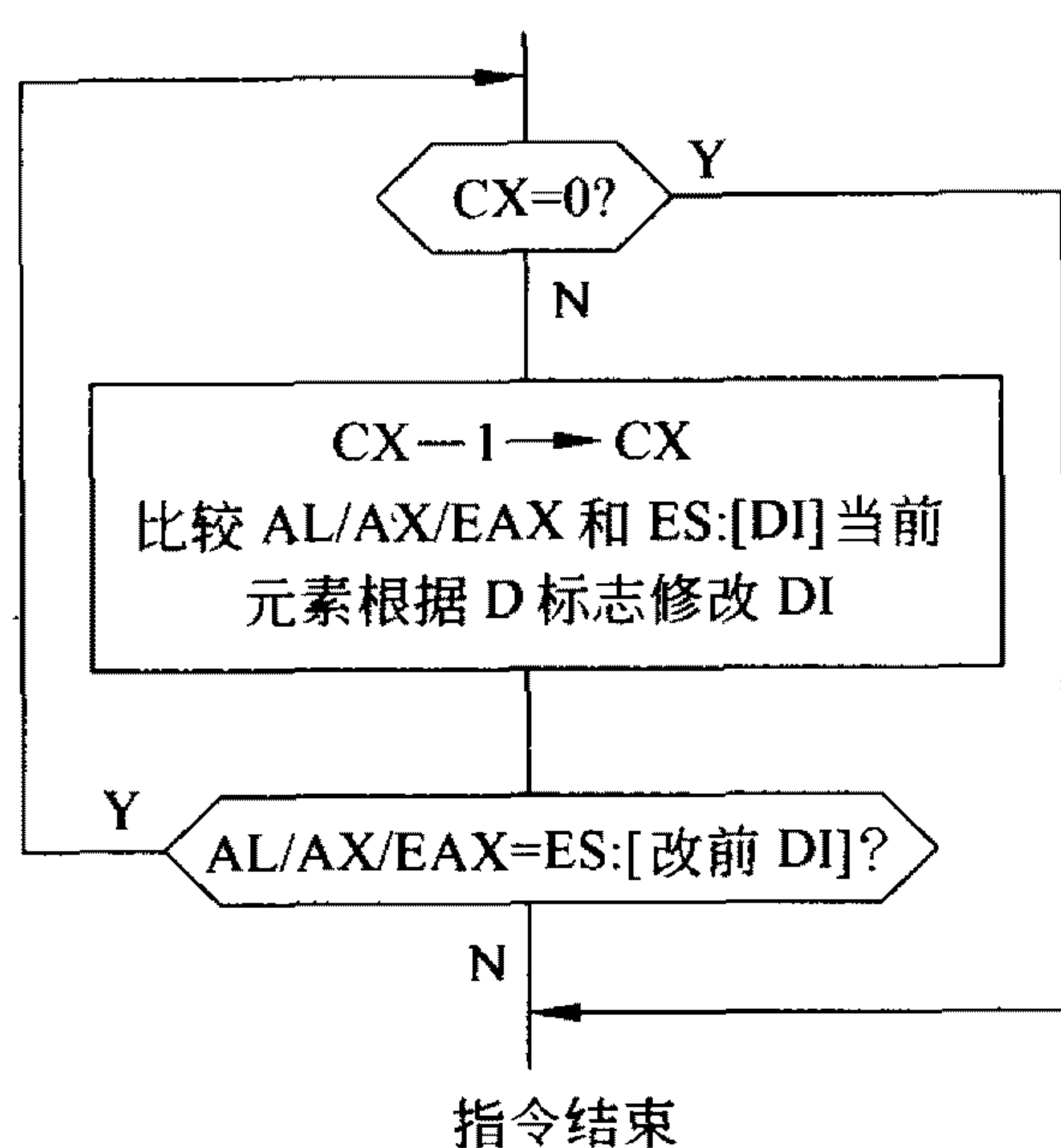
```

字节串搜索    REPNE   SCASB      (或 REPNZ SCASB)
字串搜索      REPNE   SCASW      (或 REPNZ SCASW)
双字串搜索    REPNE   SCASD      (或 REPNZ SCASD)

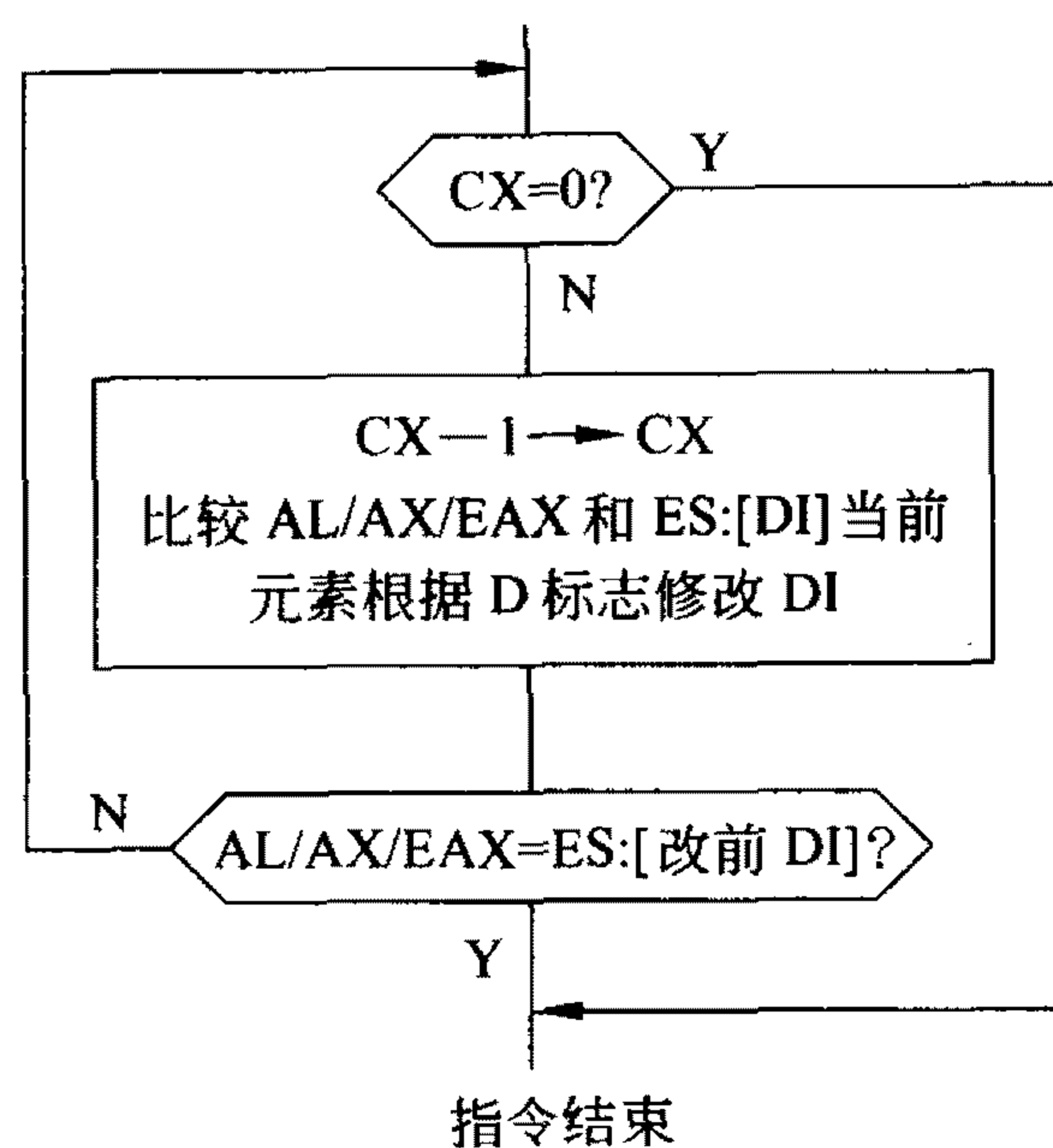
```

说明: 和基本型格式相比, 指令执行前需增加一项准备工作: 即把目标串中元素的个数→CX。该指令的功能如图 3-7(b)所示。

当预置的串长度不为 0 时, 指令执行后若 Z 标志=1, 表明目标串中至少有一个关键



REPE SCASB
(a) REPE SCASW 指令功能示意图
REPE SCASD



REPNE SCASB
(b) REPNE SCASW 指令功能示意图
REPNE SCASD

图 3-7 有重复前缀的串搜索指令功能示意图

字,若 Z 标志为 0,表示目标串中都不是要找的关键字。

3.4.6 处理器控制指令

C 标志置 0	CLC
C 标志置 1	STC
C 标志取反	CMC
D 标志置 0	CLD
D 标志置 1	STD
I 标志置 0	CLI
I 标志置 1	STI
空操作	NOP
暂停	HLT
等待	WAIT
换码	ESC
封锁	LOCK

说明:

① HLT 指令使 CPU 处于暂停状态,不执行任何操作,不影响标志。当 RESET 线上有复位信号、CPU 响应非屏蔽中断、CPU 响应可屏蔽中断,出现以上 3 种情况之一, CPU 脱离暂停状态,执行 HLT 的下一条指令。

② WAIT 指令使 CPU 处于暂停状态,直到出现外部中断为止。

③ LOCK 是指令的“前缀助记符”,可以加在任何指令的前面,CPU 执行带有 LOCK 前缀的指令后,使引脚 LOCK 输出有效信号,使系统其他设备不能占用总线。

3.5 80x86 多媒体指令

多媒体扩展(MultiMedia eXtension,MMX)指令集是 Intel 公司于 1996 在 Pentium MMX 处理器中,为增强处理器的多媒体能力提出的解决方案。MMX 是单指令流多数数据流(SIMD)指令集,允许 CPU 同时对多个数据进行并行处理,从而极大地提高了微处理器性能,使计算机能够更快速地运行图形、图像、动画、音频、视频、通信等应用程序。

数据流扩展(Streaming SIMD Extensions, SSE)是 Intel 公司于 1999 年推出 Pentium III 处理器时,针对互联网应用需求,推出的新指令集。SSE 也是一种 SIMD 指令集,可对多个浮点数进行相同的处理。

3.5.1 MMX 指令

MMX 技术是针对多媒体应用软件中经常出现的高度并行、重复执行的指令序列,为高效地处理视频、声音和图形数据而专门设计的。MMX 指令增加了 57 条多媒体指令和新的 64 位数据类型。

1. MMX 寄存器和数据格式

MMX 寄存器是随机存取的,它实际上是借用了 8 个浮点数据寄存器来实现的,如图 3-8 所示。每个浮点寄存器有 80 位,MMX 利用其低 64 位用作可随机存取的 64 位 MMX 寄存器。

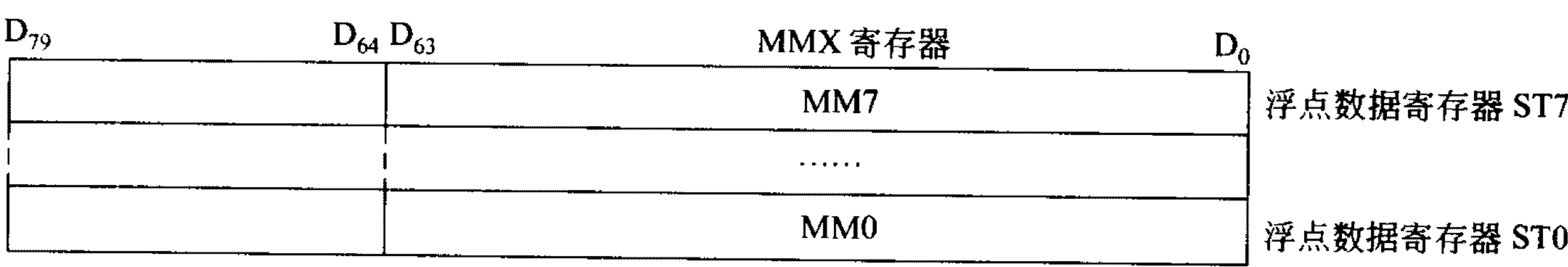


图 3-8 MMX 寄存器与浮点数据寄存器

MMX 指令主要使用紧缩整型数据,这里所说的紧缩整型数据是指多个 8/16/32 位的整型数据组合成为一个 64 位数据。它可分为四种数据类型:紧缩字节、紧缩字、紧缩双字和紧缩四字。

- ① 紧缩字节: 由 8 个字节组成一个 64 位的数据。
- ② 紧缩字: 由 4 个字组成一个 64 位的数据。
- ③ 紧缩双字节: 由 2 个双字组成一个 64 位的数据。
- ④ 紧缩四字: 一个 64 位的数据。

这样,可以通过一条 MMX 指令的使用而同时处理 8/4/2 个数据单元,即通常所说的 SIMD 结构。

2. MMX 指令格式

MMX 指令的格式和普通汇编指令格式相同,格式如下:

标号: 操作码助记符 操作数助记符 ; 注释

多数指令的操作码助记符有一个说明操作数数据类型的后缀(B、W、D、Q);如果有两个数据后缀,则第一个字母表示源操作数的数据类型,第二个字母表示目的操作数的数据类型。

例如,操作码助记符 PADD 有三种不同的操作码,记为 PADDB、PADDW、PADDD,分别表示紧缩字节、紧缩字和紧缩双字的加法。

为说明 MMX 指令允许的操作数组合,特引入如下符号:

mm : 一个 64 位 MMX 寄存器: mm0~mm7
 m64/32 : 一个 64 位或 32 位的存储器操作数
 r64/32 : 一个 64 位或 32 位的寄存器操作数
 n8 : 一个 8 位立即数

3. MMX 指令分类

MMX 指令可分为数据传送指令、算术运算指令、比较指令、类型转换指令、逻辑运算指令、移位指令、状态清除指令。

(1) 数据传送指令

- 32 位紧缩数据传送指令

格式: MOVD mm, r32/m32
 MOVD r32/m32, mm

功能: 将 32 位存储单元或寄存器的值传送给 MMX 寄存器的低 32 位,并在 MMX 寄存器的高 32 位填入 0;或将 MMX 寄存器的低 32 位传送给 32 位存储单元或寄存器。

- 64 位紧缩数据传送指令

格式: MOVQ mm, mm/m64
 MOVQ mm/m64, mm

功能: 将 MMX 寄存器或 64 位存储单元的值传送给 MMX 寄存器;或将 MMX 寄存器的 64 位数据传送给 MMX 寄存器或 64 位存储单元。

(2) 算术运算指令

MMX 运算指令实现对紧缩数据加、减、乘操作,并具有环绕、饱和以及乘加的特点。

环绕运算即通常的算术运算,当运算结果超过其数据类型所能表示的范围时,进行正常的进位和借位。

饱和运算是指运算结果超过其数据类型所能表示的范围时,其结果被最大/最小值所代替。

- 环绕加/减指令

格式: PADDB/PADDW/PADDD mm, mm/m64
 PSUBB/PSUBW/PSUBD mm, mm/m64

功能：将源操作数环绕加/减目标操作数，结果保存在目标操作数中。

- 无符号紧缩数据饱和加/减指令

格式：PADDUSB/PADDUSW mm, mm/m64

PSUBUSB/PSUBUSW mm, mm/m64

功能：将无符号源操作数饱和加/减无符号目标操作数，结果保存在目标操作数中。

- 有符号紧缩数据饱和加/减指令

格式：PADDSB/PADDSW mm, mm/m64

PSUBSB/PSUBSW mm, mm/m64

功能：将有符号源操作数饱和加/减有符号目标操作数，结果保存在目标操作数中。

- 乘法指令

格式：PMADDWD mm, mm/m64

功能：将源操作数的 4 个有符号字与目标操作数的 4 个有符号字分别相乘，结果产生 4 个有符号双字；然后，将低位的 2 个双字相加并存入目标寄存器的低位双字，高位的双字相加并存入目标寄存器的高位双字。

格式：PMULHW mm, mm/m64

PMULLW mm, mm/m64

功能：将源操作数的 4 个有符号字与目标操作数的 4 个有符号字分别相乘，结果产生 4 个有符号双字；然后，将 4 个 32 位积的高/低 16 位存入目标寄存器中，积的低/高 16 位被丢弃。

(3) 比较指令

- 紧缩数据相等比较

格式：PCMPEQB/ PCMPEQW/ PCMPEQD mm, mm/m64

- 紧缩数据大于比较

格式：PCMPGTB/ PCMPGTW/ PCMPGTD mm, mm/m64

功能：比较指令根据数据类型(B、W、D)对两个操作数的各个数据元素(字节、字、或双字)进行比较，若比较结果为真，目标寄存器相应数据元素置 1，否则置 0。

例：设 MM0=0051 0003 0045 0023H，MM1=0031 0005 0045 0023H，执行指令

PCMPEQB mm0, mm1 ; 结果 MM0=ff00 ff00 ffff ff00H

PCMPEQW mm0, mm1 ; 结果 MM0=0000 0000 ffff 0000H

PCMPEQD mm0, mm1 ; 结果 MM0=0000 0000 0000 0000H

(4) 类型转换指令

为实现各种紧缩数据的相互转换，MMX 指令中有多条紧缩和解缩指令。

- 无符号数饱和紧缩

格式：PACKUSWB mm, mm/m64

功能：将 8 个有符号紧缩字压缩成 8 个无符号字节。若有符号紧缩字大于 0FFH，则饱和处理为 0FFH；若有符号字为负数，则饱和处理为 00H。源操作数的 4 个字压缩后存入目标寄存器的低 32 位，目标操作数的 4 个字压缩后存入目标寄存器的高 32 位。

- 有符号数饱和紧缩

格式: PACKSSWB/PACKSSWD mm,mm/m64

功能: 将较大的有符号字或双字压缩成较小的有符号字节或字。若较大数据超过较小数据的上界,则饱和处理为上界;若较大数据超过较小数据的下界,则饱和处理为下界;源操作数压缩后存入目标寄存器的低 32 位,目标操作数压缩后存入目标寄存器的高 32 位。

- 高位紧缩数据解缩

格式: PUNPCKHBW/PUNPCKHBD/PUNPCKHBQ mm,mm/m64

功能: 将源操作数和目标操作数高 32 位较小的字节、字或双字交替组合成较大的字、双字或四字,操作数的低 32 位被丢弃。

- 低位紧缩数据解缩

格式: PUNPCKLBW/PUNPCKLBD/PUNPCKLBQ mm,mm/m64

功能: 将源操作数和目标操作数低 32 位较小的字节、字或双字交替组合成较大的字、双字或四字,操作数的高 32 位被丢弃。

(5) 逻辑运算指令

格式: PAND/PANDN/POR/PXOR mm,mm/m64

功能: 将源操作数和目标操作数按位进行逻辑与、与非、或、异或操作,结果保存在目标操作数中。

(6) 移位指令

- 紧缩逻辑左移

格式: PSLLW/PSLLD/PSLLQ mm,mm/m64/n8

功能: 以字、双字或四字为数据元素单位,按源操作数的数值左移目标操作数中的数据元素,低位用 0 填充。

例: 设 MM0=0051 0003 0045 0023H,执行指令

PSLLW mm0,4 ;执行结果 MM0=0510 0030 0450 0230

- 紧缩逻辑右移

格式: PSRLW/PSRLD/PSRLQ mm,mm/m64/n8

功能: 以字、双字或四字为数据元素单位,按源操作数的数值右移目标操作数中的数据元素,高位用 0 填充。

- 紧缩算术右移

格式: PSRAW/ PSRAD mm,mm/m64/n8

功能: 以字、双字为数据元素单位,按源操作数的数值右移目标操作数中的数据元素,高位用该数据元素原来的符号位填充。

(7) 状态清除指令

格式: EMMS

功能: 清空浮点数据寄存器,使后续浮点指令可以使用浮点寄存器。

4. MMX 指令程序设计

MASM 6.11 以上版本就可支持 MMX 指令的汇编编程。采用 MMX 指令的汇编语言编程,与普通汇编语言编程基本相同,但要注意以下几点:

(1) 检查 CPU 是否支持 MMX 指令

源程序一开始应使用 .586 和 .MMX 方式选择伪指令。因此程序只能在 Pentium MMX 以上处理器的计算机运行。

另外在程序开始前应首先检查 CPU 是否支持 MMX 指令集。检查方法为:在 EAX=1 时,执行 CPUID 指令,若返回 EDX 的 $D_{23}=1$ 表示 CPU 支持 MMX 指令。代码如下:

```

MOV EAX,1
CPUID          ;Pentium MMX 新增指令,用于判断 MMX 处理器是否存在
TEST EDX,00800000H
JNZ MMX_OK     ;D23=1 表示处理器支持 MMX 功能,
               ;处理器不支持 MMX,采用整数指令完成功能
.....
JMP EXIT
MMX_OK:;处理器支持 MMX,采用 MMX 指令完成功能
.....
EXIT:         ;程序结束

```

(2) MMX 指令与浮点指令的混合使用

由于 MMX 寄存器是使用浮点数据寄存器实现的,程序不能同时在一个寄存器中既使用浮点数据又使用紧缩数据。浮点指令和 MMX 指令之间的切换需一定的时间,不要在应用程序中混合使用浮点指令与 MMX 指令。

避免的方法:MMX 程序段和浮点程序段分别独自放在各自的代码段中;退出 MMX 指令前要用 EMMS 指令清空 MMX 状态,否则将引起数据溢出。同样,在退出浮点指令前,也应清空浮点数据寄存器栈;在进行任务切换时,不要用 MMX 或浮点寄存器传递参数。

(3) MMX 程序的优化

用 MMX 指令可提高程序性能,程序应合理安排指令,使多条指令尽可能在 CPU 内部并行执行。

【例 3.5.1】 4 元素的向量点积运算。

【程序分析】

向量 a_i 和 b_i ($i=1, \dots, n$) 的点积 $a \cdot b = a_1 \times b_1 + a_2 \times b_2 + \dots + a_n \times b_n$, 其中 a_i 和 b_i 的长度为 16 位,一个字;积的长度为 32 位,一个双字。

由于 MMX 指令 pmaddwd 可同时处理 4 个 16 位紧缩字分别相乘和 4 个 32 位积的相加,因此采用该指令可大大提高运算速度。

【程序清单】

. MMX

数据段

a DQ 0002000300040005h ;定义 a[i]的 4 个元素

b DQ 0006000700080009h ;定义 b[i]的 4 个元素

result DD ?

代码段

.....

MMX_OK:

MOVQ MM0,a ;取数据 a[i]的 4 个元素

PMADDWD MM0,b ;数据 a[i]的 4 个元素与 b[i]的 4 个元素相乘,然后相加

MOVQ MM1,MM0

PSRLQ MM1,32

PADDD MM0,MM1 ;将高位的两个双字右移 32 位+低位的两个双字

MOVD result,MM0 ;保存结果

EMMS ;退出 MMX 指令段

3.5.2 SSE 指令

SSE 指令集是 Intel 在 Pentium III 处理器中率先推出的。SSE 技术支持 128 位紧缩浮点数据,提供 8 个 SIMD 浮点数据寄存器 XMM0~XMM7。SSE 兼容 MMX 指令,它可以通过 SIMD 和单时钟周期并行处理多个浮点数据来有效地提高浮点运算速度。SSE 指令对目前流行的图像处理、浮点运算、3D 运算、视频处理、音频处理等诸多多媒体应用起到全面强化的作用。

Intel 后来又在 SSE 的基础上开发了 SSE2,增加了一些指令,使得其 Pentium 4 处理器性能大大提高。到 Pentium 4 设计结束为止,Intel 增加了一套包括 144 条新建指令的 SSE2 指令集。SSE2 能处理 128 位和两倍精密浮点数学运算。处理更精确浮点数的能力使 SSE2 成为加速多媒体程序、3D 处理工程以及工作站类型任务的基础配置。

SSE 指令集包括了 70 条指令,其中包含提高 3D 图形运算效率的 50 条 SIMD 浮点运算指令、12 条 MMX 整数运算增强指令和 8 条高速缓冲存储器优化指令。

(1) SIMD 浮点运算指令

包括数据传送指令、算术运算指令、逻辑运算指令、比较指令、类型转换指令、组合指令和状态管理指令。

(2) MMX 整数运算增强指令

这是为了增强和完善 MMX 指令系统而新增加的指令,是原 MMX 指令的扩展。

(3) 高速缓冲存储器优化指令

为了更好地控制 Cache 的操作,提高程序运行性能,SSE 技术设计了 8 条高速缓冲存储器优化指令。

目前,SSE 指令的编程主要是采用在高级语言(如 Visual C++)中,通过内嵌汇编的

方法来实现,其目的是提高程序的并行数据处理能力。在用 SSE 指令之前也必须首先检查 CPU 是否支持 SSE 指令集。检查方法为:在 $EAX=1$ 时,执行 CUID 指令,若返回 EDX 的 $D_{23}=1$ 表示 CPU 支持 SSE 指令。由于本书篇幅所限,SSE 指令程序的具体实现,请读者参考其他资料。

习 题

1. 写出下列用逻辑地址表示的存储单元的物理地址

(1) 1234H; 5678H (2) 2F34H; 2F6H

(3) 576AH; 1024H (4) 2FD0H; 100H

2. 列表写出下列指令中目标操作数、源操作数的寻址方式,如果有非法的内存操作数请改正,并写出 CPU 所寻址的逻辑段。

```
(1) MOV     BX,50
```

(2) CMP [BX], 100

(3) ADD [SI], 1000

```
(4) MOV     BP,SP
```

```
(5) MOV     BX,[BP+4]
```

```
(6) MOV     AX,[BX+DI+5]
```

3. 以 2^{16} 为模, 将 C678H 分别和下列各数相加, 列表写出十六进制和数, 以及 A、C、O、P、S、Z 6 种状态标志的值。

(1) CF23H

(2) 6398H

(3) 94FBH

(4) 65E2H

4. 分别用一条指令完成:

(1) AH 高 4 位置 1, 低 4 位不变。

(2) BH 高 4 位取反,低 4 位不变,BL 高 4 位不变,低 4 位取反。

(3) CX 低 4 位清 0, 其他位不变。

5. 已知数据段有:

FIRST DB 12H,34H

SECOND DB 56H,78H

- ① 要求采用传送指令编一段程序,实现:

FIRST 和 SECOND 单元的内容互换, FIRST+1 和 SECOND+1 单元的内容互换。

- ② 设 $SS=2000H$, $SP=3456H$, 用堆栈指令编一段程序实现上述要求, 并画出堆栈空间的数据变化示意图。

6. 下列程序段执行后 AX=?

设数据段有：

TABLE	DW	158,258,358,458
-------	----	-----------------

ENTRY DW 3

代码段 对 DS 初始化

MOV BX,OFFSET TABLE

MOV SI,ENTRY

MOV AX,[BX+SI]

7. 运用除法指令完成: $1.193182\text{MHz} \div 433\text{Hz}$, 将商值送入数据段 XX 字单元。

8. 把 AH 低四位和 AL 低四位拼装成一个字节(AH 低四位为拼装后的高四位) \rightarrow AH。

9. 将 AL 中的 8 位二进制数,按倒序方式重新排列,即 AL 原来为 $D_7 D_6 \dots D_0$,倒序后 $AL = D_0 D_1 \dots D_7$ 。

10. 设数据段有:

BUF DB 50 DUP(?) ;50 个有符号数

分别编写四个程序段:

① 将其中的正数送数据段 PLUS 开始的若干单元。

将其中的负数送数据段 MINUS 开始的若干单元。

② 将其中的非零的数送数据段 NOT0 开始的若干单元。

③ 分别求出它们的绝对值。

④ 设 BUF~BUF+3 四个单元存放的是一个双字型有符号数(BUF 单元为最低 8 位)求出它的绝对值。

11. 下列程序段执行后 AL=?

MOV AL,78H

STC

DEC AL

DAS

12. 设 AL 中为合法的组合 BCD 码,下列程序段的功能是什么?

SHR AL,1

TEST AL,8

JZ DONE

SUB AL,3

DONE:.....

13. 把内存 12345H 开始的 1KB 转送到 23456H 开始的内存区。

14. MMX 指令系统和 SSE 指令系统有哪些特点?

15. MMX 指令系统支持哪些数据类型?

16. 用 MMX 指令完成: 8 个字数据数组的加、减、乘、除运算。

宏汇编语言

众所周知,计算机硬件只能识别运行用 0、1 代码书写的指令,但是用机器指令编程其困难程度是可想而知的,于是专家们发明了汇编语言。汇编语言是面向机器的编程语言,随着 CPU 的更新换代,指令系统的功能逐步提高,汇编语言的版本也不断升级。本章仅介绍宏汇编语言中常用的语句和伪指令。

4.1 汇编语言程序的开发过程

汇编语言程序要经过编辑、汇编、链接才能生成可执行文件。

1. 源程序的编辑

编辑就是调用编辑程序(如 EDIT. EXE 或 QEDIT. EXE)生成一个扩展名为 ASM 的汇编源文件,如果是对原有的 ASM 文件进行修改,那么修改前的 ASM 文件主文件名不变,扩展名自动改为 BAK,修改后的文件扩展名为 ASM。

2. 源程序的汇编

汇编就是调用汇编程序(如 MASM. EXE 或 TASM. EXE)对源程序(一定要有扩展名 ASM)进行翻译,汇编的主要目的是生成扩展名为 OBJ 的目标文件,如果源程序有语法错误汇编后不生成目标文件,并且给出错误信息。

Borland 公司开发的宏汇编程序 TASM. EXE 要求用户使用命令行方式(不是会话方式)给出汇编信息,我们假设 TASM. EXE 以及下面讲到的链接程序 TLINK. EXE 和待汇编的源文件(假设文件名为 ABC. ASM)都在 C 盘,则汇编时简易的命令行格式如下:

```
C: >TASM/汇编参数    ABC
```

如果源程序没有语法错误,汇编后在 C 盘上可以得到同名的目标文件 ABC. OBJ,如果仅仅键入“TASM”,则屏幕显示出若干个汇编参数及其功能,其中“/L”的作用是汇编后同时生成同名的列表文件(扩展名为 LST)。命令行中“/汇编参数”不是必需的。

3. 目标程序的链接

链接就是调用链接程序(如 LINK. EXE 或 TLINK. EXE)对目标文件和库文件进行链接,生成扩展名为 EXE 或 COM 的可执行文件。需要注意的是,即便程序设计中没有库文件(本书的所有例题都没有使用库文件),或者仅仅是单一模块的目标文件,也必须经过“链接”这一步骤,才能生成可执行文件。

Borland 公司的链接程序 TLINK. EXE 也要求用户使用命令行方式给出链接信息,假设链接的对象是 C 盘上的 ABC. OBJ,那么链接时,简易的命令行格式如下:

C: >TLINK/链接参数 ABC

如果仅仅键入“TLINK”,则屏幕显示出若干个链接参数及其功能,其中“/ t”表示链接后生成的可执行文件是同名的 COM 文件(当然,源程序必须符合 COM 文件的编程格式),没有“/ t”,生成的可执行文件是同名的 EXE 文件。

说明:链接程序 TLINK. EXE 还需要有两个支撑文件,它们是: DPMI16BI. OVL 和 RTM. EXE。

建议读者自备一张“Borland Turbo Assembler 5.0”光盘,执行其中的“INSTALL. EXE”文件,根据操作提示将相关文件解压缩存入硬盘备份,再在 C 盘建立一个“TASM”子目录,将 TASM. EXE, TLINK. EXE, DPMI16BI. OVI, RTM. EXE,外加一个编辑文件存放到 TASM 子目录下,从而构成一个集编辑、汇编、链接为一体的开发环境。

4.2 汇编源程序的语句类型

汇编源程序采用分段结构,汇编程序对源程序是以语句为单位进行汇编的,一个完整的汇编源程序至少应包含两类语句:

一类是指令性语句,即通常所说的符号指令;另一类是指示性语句,即伪指令。

指令和伪指令是两种不同的概念。指令是通知 CPU 进行某种操作的命令,由硬件完成其功能。伪指令为汇编程序提供汇编信息,为链接程序提供链接信息,显然伪指令的功能是由相应的软件完成的。

指令性语句的书写格式如下:

标号: 符号指令 ;注释

它由三部分组成:标号及其后的冒号、符号指令、分号及其后的注释。标号不是必需的,只有当某条指令作为转移指令的目标时,该条指令才需加上标号,标号和指令之间必须用冒号间隔。注释可有可无,CPU 不执行,只是在列出程序清单时,照原样显示,便于阅读。

指示性语句的书写格式如下:

变量名 伪指令 ;注释

变量名也不是必需的,一条指示性语句,如果有变量名的话,则变量名与伪指令之间用“空格”做间隔符,这是指示性语句与指令性语句在书写格式上的区别。

宏汇编语言由于版本不同,伪指令的种类也略有不同,其中有些是程序设计中必须用到或经常用到的,称为基本伪指令、基本语句,有些伪指令很少使用,有些伪操作并不能简化程序设计,这里就不一一介绍了。

4.3 宏汇编基本语法

4.3.1 标号、变量和常量

1. 标号和变量

标号和变量的命名规则是:以字母或者下划线开头,后跟字母、数字、下划线,长度不超过 31 个字符,但系统保留字不能做为标号和变量名。保留字是系统专用的有特殊意义的名字,例如:指令的操作码助记符、运算符、寄存器名称、伪指令助记符等。

标号代表指令地址,它为转移指令提供了转移目标。变量代表内存操作数的存储地址,或者说变量名就代表某个单元。由于标号和变量是用一串字符命名的,从这个意义上讲,标号和变量又称为符号地址。

标号被定义在代码段,变量通常被定义在数据段、附加段或堆栈段,故而标号和变量都有三个属性。它们是:

(1) 段属性

标号或变量所在段的段基址,用 SEG 运算符可以算出。

(2) 偏移属性

标号或变量所代表的单元,相对于段首址之间的地址偏移量(又称有效地址),用 OFFSET 运算符可以求出。

(3) 类型属性

① 变量的类型有字节型、字型、双字型、四字型等。

用 DB 伪指令定义的变量,其所属的单元均为字节型。用 DW、DD、DQ 伪指令定义的变量,其所属的单元分别有:字型、双字型和四字型属性。

了解变量的类型是十分重要的,汇编语言规定:读写内存操作数时源、目标操作数的类型必须一致。变量的类型可以用 PTR 运算符做临时性的修改。

② 标号的类型有 FAR(远)、NEAR(近)两种属性。

如果某个标号是段内转移指令的目标地址,那么这个标号是近程标号,它的类型属性为 NEAR(近),如果某个标号是其他代码段转移指令的目标地址,那么这个标号的类型属性为 FAR(远)。

2. 常量

常量包括立即数、字符串常数和符号常数。

(1) 立即数

如:12,0A8H,10101010B,34Q,-2

立即数必须以数字开头,以 A~F 开头的十六进制数必须加前缀数字 0。立即数的数制用后缀表示,后缀 D(或者缺省)表示十进制数,后缀 H 表示十六进制数,后缀 B 表示二进制数,后缀 Q 表示八进制数。程序员可以按自己的习惯书写立即数,经过汇编之后,汇编程序将把各种进制表示的立即数转换成等值的二进制数,负数转换成补码。

(2) 字符串常数

用单引号括起来的字符串称为字符串常数。如 'A', 'P10'……经过汇编之后,单引号中的每个字符转换成相应的 ASCII 码,可以像使用立即数一样使用它们,例如:

```
MOV    DL,'1'    ;31H → DL
```

(3) 符号常数

符号常数用伪指令“EQU”或者伪指令“=”定义,使用符号常数有利于程序调试,增加程序的可读性,符号常数经过定义之后,也可以像立即数一样使用,例如:

```
COUNT    EQU    55
POINTER   =      2F8H
.....
MOV    CL,COUNT    ;CL=55
MOV    DX,POINTER  ;DX=2F8H
```

4.3.2 运算符

1. 数值运算符

① 算术运算符有: + (加), - (减), * (乘), / (除), MOD (模除) 5 种。其中,“模除”的概念是:做除法取其余数,如:

```
123    MOD    45    ;结果是 33
```

算术表达式可以作为立即数,它们是在汇编时由汇编程序完成运算的,如果表达式的值超出范围,汇编时给出错误信息,如:

```
MOV    AX,22 * 33    ;正确
MOV    AL,22 * 33    ;错误
```

② 逻辑运算符有: NOT (非), AND (与), OR (或), XOR (异或), HIGH, LOW。其中 HIGH, LOW 为分离运算符, HIGH 截取操作数的高 8 位, LOW 截取操作数的低 8 位,如:

```
X    EQU    11110000B
.....
MOV    AL,X AND 01010101B    ;AL=50H
MOV    AX,HIGH 5566H        ;AX=55H
```

③ 关系运算符有: EQ (等于), NE (不等于), GT (大于), LT (小于), GE (大于等于), LE (小于等于)。

2. 修改属性的运算符

前文讲过,标号和变量都有类型属性,后面还要讲到用 DB 伪指令定义的变量是字节型变量,用 DW、DD 定义的变量分别是字型和双字型变量。在程序设计中经常遇到这样的问题:某个变量的类型已经定义了,在访问这些变量的时候,为了保证操作数类型匹配,需要改变变量的类型怎么办?为此汇编语言提供了 PTR 运算符。PTR 运算符的使用格式如下:

类型说明符 PTR 地址表达式

其中类型说明符有:BYTE(字节),WORD(字),DWORD(双字),FAR(远),NEAR(近)。地址表达式可以是转移地址标号、过程名或内存操作数的 5 种寻址方式之一。

PTR 运算符只能出现在指令的操作数部分,它的功能是在本条指令中临时修改由地址表达式指向的内存单元、转移地址标号或子程序的属性。

书写指令时,遇到下列情况之一,必须用 PTR 运算符修改或确认内存操作数的类型:

① 在双操作数指令中(如:MOV、ADD、CMP、AND...),源为立即数,目标为直接寻址的内存操作数,但源、目标类型不一致;源为单字节或双字节立即数,目标是采用间址、基址、变址或基址加变址寻址的内存操作数,这两种情况必须用 PTR 运算符临时修改内存操作数的类型。

② 在单操作数指令中(如:INC、DEC、NEG...),操作数为非直接寻址的内存操作数,则内存操作数必须用 PTR 运算符明确说明其类型。

3. 返回属性或数值的运算符

汇编语言中有一些单目运算符,它们加在运算对象之前,可求出对象的某个参数。常用的有以下几个:

(1) SEG 运算符

格式:SEG 段名

功能:计算某个逻辑段的段基址。

例如:

```
MOV    AX,SEG DATA
MOV    DS,AX
```

算出段名为 DATA 的逻辑段段基址并赋给 AX,再转赋给 DS 寄存器。用 SEG 运算符,也可以求出段中某个变量或标号的段基址。

(2) OFFSET 运算符

格式:OFFSET 变量名或标号名

功能:算出变量名(或标号名)所在单元的偏移地址。

例如:假设数据段:

```
BUF    DB      12,34,56
```

```
...  
MOV    AX,SEG 数据段段名  
MOV    DS,AX  
MOV    BX,OFFSET BUF  
MOV    AL,[BX]  
.....
```

算出变量 BUF 所在单元的偏移地址并赋给 BX,对数据段用 BX 间址取数送 AL,所以 AL=12。

(3) TYPE 运算符

格式: TYPE 变量名或标号

功能: 算出变量或标号的类型,对于字节型变量,返回值为 1,字型变量返回值为 2,双字型变量返回值为 4。

(4) \$ 运算符

\$ 运算符返回汇编计数器的当前值。具体用法见“字节定义伪指令”。

4. 方括号运算符和地址表达式

用方括号括起来的地址表达式,是访问内存操作数常用的寻址方式。方括号的另一个用途是表示数组的下标,下标可以是常数、算术表达式、16 位或 32 位的寻址方式表达式。用下标访问数组元素属于直接寻址范畴。

4.4 数据定义伪指令

伪指令、伪语句是汇编语言提供的指示性语句,它为汇编程序和链接程序提供信息。伪指令、伪语句本身不占用内存单元,它们的功能是在汇编和链接时由相应的软件完成的。本节仅介绍最常用的伪指令。

1. 等值伪指令

格式: 符号常数 EQU 表达式

功能: 将表达式的值赋给一个符号常数。

例如:

```
XX      EQU      12  
COUNT  EQU      55
```

2. 等号伪指令

格式: 符号常数=表达式

功能: 将表达式的值赋给一个符号常数。

例如:


```
XX      = 12
COUNT = 55
```

说明：用 EQU 伪指令定义的符号常数，其值在后续语句中不能更改，用等号伪指令定义的符号常数，在后续语句中可以重新定义。符号常数一经定义，在后续语句中就可以像立即数一样使用它们。定义语句可以放在任何逻辑段。

3. 字节定义伪指令

格式：变量名 DB 一串用逗号间隔的单字节数

例如：

```
BUF1      DB      22H,5*6,10101010B
           DB      120,-5,0A6H,'HELLO'
COUNT    EQU     $-BUF1
BUF2      DB      ?,?,10 DUP('A')
```

功能与说明：

- ① DB 是 Define Byte 的缩写，? 代表随机数。
- ② DUP 是 Duplicate 的缩写，译为“重复”，左边是重复系数，右边圆括号中为需要重复设置的数据，10 DUP('A')等价于 10 个用逗号间隔的 41H。
- ③ \$ 代表汇编计数器的当前值，最常见的用法是和等值、等号伪指令配合，紧跟在 DB、DW... 伪指令之后，统计分配给某个变量的单元数，本例 COUNT 等于 11。
- ④ DB 伪指令的功能是：通知汇编程序，把所定义的单字节数转换成二进制数，并从指定的变量单元开始依次存放。用 DB 伪指令定义的这些单元的属性为“字节型”。

4. 字定义伪指令

格式：变量名 DW 一串用逗号间隔的双字节数

例如：

```
WNUM      DW      1234H,56,'AB','C',?
```

功能与说明：

- ① DW 是 Define Word 的缩写，用 DW 定义的问号为双字节随机数。
 - ② 出现在 DW 伪指令中的字符串常数，单引号中只能是一个或两个字符。
 - ③ DW 伪指令通知汇编程序，把所定义的双字节数从指定变量开始依次存放，每一个双字节数的存放规律是：低位字节存入低地址单元(i 单元)，高位字节存入高地址单元(i+1 单元)。
- 上例经汇编之后，数的存放规律是：WNUM~WNUM+9 单元依次为 34H、12H、38H、00H、42H、41H、43H、00H、××、××。
- ④ 被 DW 定义的这些单元的属性为“字型”。

5. 双字定义伪指令

格式：变量名 DD 一串用逗号间隔的 4 字节数

例如：

```
DNUM    DD    ?,12345678H
```

功能：通知汇编程序，把 DD 定义的数，从指定的变量名开始依次存放，每一个数占 4 个单元，每一个数的存放规律也是低位字节存入低地址单元，较高字节存入较高的地址单元。被 DD 定义的这些单元都有“双字型”属性。

6. 多字节定义伪指令

格式：变量名 DF/DQ/DT 一串用逗号间隔的多字节数

说明：这是高版本汇编程序新增的伪指令，伪指令助记符分别为 DF、DQ、DT，它们分别为所定义的每一个数，分配 6 个、8 个、10 个单元。

为了深入理解数值定义伪指令、学习 PTR 运算符的使用，请仔细阅读以下例题。

【例 4.4.1】 阅读以下程序，写出指令执行后的目标操作数。

设数据段：

```
BNUM    DB    12H,34H,56H,78H,90H
WNUM     DW    1122H,3344H,5566H
DNUM     DD    13572468H,87654321H
FNUM     DF    112233445566H
```

代码段：……

```
MOV  AX,SEG 数据段段名
MOV  DS,AX                      ;对 DS 初始化
MOV  BL,BNUM                     ;BL=12H
MOV  BX,WNUM+2                   ;BX=3344H
MOV  EBX,DNUM+4                  ;EBX=87654321H
MOV  BL,BYTE PTR DNUM            ;BL=68H
MOV  BX,WORD PTR BNUM+1          ;BX=5634H
MOV  EBX,DWORD PTR WNUM+1        ;EBX=66334411H
MOV  BL,BNUM[2*2]                ;BL=90H
MOV  BX,WNUM[4]                  ;BX=5566H
MOV  EBX,DNUM[3]                 ;EBX=65432113H
MOV  BX,WORD PTR DNUM[3]         ;BX=2113H
MOV  EBX,DWORD PTR BNUM[0]       ;EBX=78563412H
MOV  BX,WORD PTR FNUM            ;BX=5566H
MOV  BX,3
MOV  BL,BNUM[BX]                 ;BL=78H
MOV  SI,OFFSET BNUM
MOV  BX,[SI+1]                   ;BX=5634H
```

以上指令如果删掉 PTR 以及相应的类型说明符，则该指令即为非法指令。

4.5 宏汇编语言基本语句

这类语句是与程序结构密切相关的基本语句,用来说明 CPU 类型、段结构与寻址方式的段约定、目标块的定位和源程序结束。

汇编语言要求,一个完整的源程序在结构上必须做到:

用方式选择伪指令说明执行该程序的 CPU 类型;

用段定义语句定义每一个逻辑段;

用过程定义语句定义每一个子程序;

用 ASSUME 语句说明段约定;

用汇编结束语句说明源程序结束。

此外,程序在完成预定功能之后,应能安全返回 DOS。

1. 方式选择伪指令

80586 指令集是在 8086、8088、80286、80386、80486 基础上逐步发展起来的,很显然,80586 的某些指令在早期的 80x86 系列是没有的,为了使汇编程序能够识别,在高版本的汇编程序中,对应于每一种 CPU 的指令系统,都有一个目标指令的集合。方式选择伪指令通知汇编程序,当前的源程序指令是哪一种 CPU 指令,经过汇编链接之后生成的目标程序在哪一种 CPU 机型上运行,不属于选定 CPU 的指令均为非法指令。方式选择伪指令以句号开头,其格式和功能描述如下:

. 8086	;只汇编 8086、8088 指令。
. 286 或. 286C	;只汇编 8086、8088 及 80286 实地址模式指令
. 286P	;只汇编 8086、8088 及 80286 全部指令
. 386 或. 386C	;同. 286,且汇编 80386 实地址模式指令
. 386P	;同. 286P,且汇编 80386 全部指令
. 486 或. 486C	;同. 386,且汇编 80486 实地址模式指令
. 486P	;同. 386P,且汇编 80486 全部指令
. 586 或. 586C	;同. 486,且汇编 80586 实地址模式指令
. 586P	;同. 486P,且汇编 80586 全部指令

通常,方式选择伪指令放在程序的头部,做为源程序的第一条语句。不设置方式选择伪指令与设置. 8086 是等价的。

2. 段定义语句

段定义语句是逻辑段的定界语句,源程序中每一个逻辑段都必须用段定义语句定界。段定义语句格式如下:

```

段名  SEGMENT    定位参数    链接参数    '分类名'    段长度
      段体
段名  ENDS

```


SEGMENT/ENDS 是一对段定义语句,一个逻辑段从 SEGMENT 语句开始,到 ENDS 语句结束。段名的命名规则和变量名以及标号名一样,段名不能代表段体的性质,但是为了阅读方便,习惯上总是根据段体的性质起一个适当的段名。通常用 DATA 做为数据段的段名,用 STACK 做为堆栈段的段名,CODE 为代码段的段名。

定位参数,链接参数,'分类名'是段定义语句的 3 个属性参数,可以选用 1~3 个,也可以全部省略。

段定义语句的属性参数为源程序的汇编、链接提供必要的信息,特别是模块化程序,各个模块如何定位,彼此之间如何链接,将较多的涉及定位参数和链接参数的选择,因此预先了解一点模块化程序的基本知识是必要的。

为了加快程序开发的速度,通常把一个大型的程序,分解成若干个有独立功能的小程序。能独立汇编的逻辑段称为一个“模块”。每个模块都可以有自己的数据段、代码段……各个模块单独编辑,单独汇编,生成各自的 OBJ 文件,然后通过链接程序将各个 OBJ 文件链接起来,生成一个 EXE 文件。链接顺序可以是:

OBJ1 + OBJ2 + OBJ3 + ……

也可以是:

OBJ3 + OBJ2 + OBJ1 + ……

不同模块之间,同名段如何链接? 如何定位? 反过来讲,定位参数,链接参数如何描述才能达到设计者的目的呢? 下面将详细介绍段参数的作用。

(1) 定位参数

定位参数通知链接程序,逻辑段的目标代码在存储器中如何存放。定位参数有 4 种描述方式可供选择:

- ① BYTE 代表字节地址:表明该逻辑段的目标代码可以从任意地址开始依次存放。
- ② WORD 代表字地址:表示该逻辑段的目标代码,从偶地址开始依次存放。
- ③ PARA(或者缺省)代表节地址:表示该逻辑段的目标代码,从一个能被 16 整除的地址开始依次存放。
- ④ PAGE 代表页地址:表示该逻辑段的目标代码,从一个能被 256 整除的地址开始依次存放。

链接程序对于不同模块中的同名段进行链接时,对于有 BYTE 属性的段,总是紧接着前一段存放,不留空闲单元。对于有 WORD 属性的段,也是紧接前一段存放,最多留出一个空闲单元。

(2) 链接参数

链接又称组合,链接参数有 6 种描述方式可供选择。

- ① PUBLIC 它通知链接程序,把不同模块中具有 PUBLIC 属性的同名段,在满足定位方式前提下,按照指定的链接顺序进行链接,组成一个逻辑段。
- ② MEMORY 实验证明 MEMORY 属性和 PUBLIC 属性是等价的。
- ③ COMMON 它通知链接程序,把不同模块中,具有 COMMON 属性的同名段,根据指定的链接顺序,按照“覆盖”方式组合成一个逻辑段。组合之后的逻辑段体积,等于链接之前具有 COMMON 属性同名段中最大的段的体积。

④ STACK 具有 STACK 属性的逻辑段是堆栈段,链接程序将把不同模块中具有 STACK 属性的同名段链接成一个大的堆栈段。链接后的堆栈空间是链接前各模块预留的堆栈空间之和。

程序装入后,DOS 自动给 SS 寄存器赋值,使之等于堆栈段段基址,自动给 SP 赋值,使之等于堆栈空间的字节数,使 SS: SP 自动指向栈顶。

链接程序要求: EXE 文件的汇编源程序,必须有堆栈段,否则链接时发出警告信息:

Warning no stack segment

这行信息仅仅是提醒用户注意,并不表示源程序有什么错误。

⑤ AT 表达式。

该属性表明:逻辑段在定位时,其段基址等于表达式给出的值。AT 属性常和 ORG 伪指令配合,例如:

```
DATA      SEGMENT AT 0040H
          ORG      0017H
KEYFLAG   DB      ?
DATA      ENDS
```

它定义键标志单元(变量 KEYFLAG)的物理地址为 00417H,以便代码段中可以使用变量名 KEYFLAG 访问这个单元。

AT 参数不能使用在代码段。实验表明:在用 AT 参数描述的逻辑段中,企图使用 DB,DW 等数值伪指令预置数据是不会成功的。

⑥ 缺省:表明该段是一个独立的逻辑段,链接程序对于不同模块中,链接参数缺省的同名段,不进行组合。

图 4-1 形象地画出了不同模块之间同名段的 3 个链接参数的作用,链接顺序为:主模块 OBJ+子模块 OBJ。

(3) '分类名'

分类名表示逻辑段的类别,分类名是用户定义的,长度不超过 40 个字符的字符串,分类名必须用单引号括起来,分类名可有可无。

习惯上,数据段分类名用'DATA',代码段分类名用'CODE',堆栈段分类名用'STACK'。

链接程序把不同模块中分类名相同的同名段组织成一类,存放在邻近的存储区中。

(4) 段长度

这一参数是 80386 以后汇编语言新增的段参数,只有高版本的汇编器才能识别,它有两种描述方式可供选择:

① USE16:表示该逻辑段长度最大允许为 64KB,单元的偏移地址为 16 位,访问该逻辑段应采用 16 位寻址方式。

② USE32:表示该逻辑段长度可以超过 64KB,单元的偏移地址为 32 位,访问该逻辑段采用 32 位寻址方式。

再次重申:运行于实地址模式下的程序若访问高于 64KB 的单元(其偏移地址必然大于 FFFFH)必将引起系统异常中断,导致死机。例如:

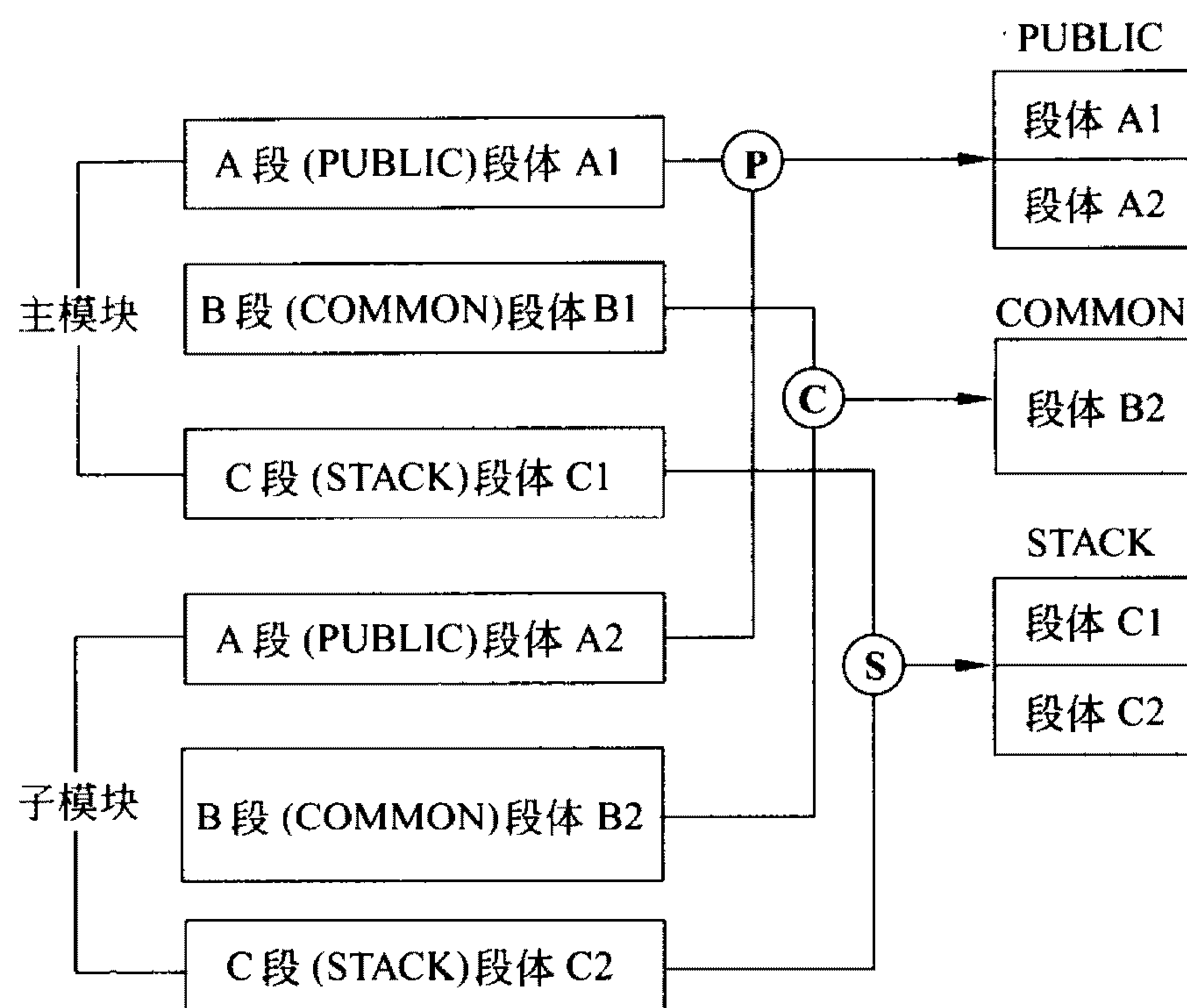


图 4-1 不同模块同名段链接参数的功能示意图

```

FDATA    SEGMENT USE32
          DB      65536 DUP(?)
FBUF     DB      'A'
FDATA    ENDS

```

代码段若执行：MOV AL,FBUF 系统立即瘫痪！

DOS 环境就是实地址模式环境,在实地址模式环境下运行的程序其代码段应当选用 USE16 段长度,否则有的程序就不能正常运行。段长度这一参数可以缺省,但缺省之后默认的段长度在不同版本的汇编器上有不同的解释。

在方式选择伪指令选用 .386 或 .486 的前提下:

Turbo Assumbler 4.1 版本,默认的段长度为 USE32。

Turbo Assumbler 5.1 版本,默认的段长度为 USE16。

有鉴于此,我们在以后的程序设计例题中,就段长度而言,一律不采用缺省方式,而是不厌其烦地明确写出 USE16 或 USE32,这样做便于阅读,而且不受汇编器版本的限制。

【小结】

段定义语句有 4 个属性参数,每一个参数都有多种选择,通常只在模块化程序中才有必要仔细考虑各模块之间同名段的定位方式和链接方式,对于单一模块的程序没有必要考虑这些问题。

对于单一模块的程序,如果有堆栈段的话,堆栈段的链接属性选用 STACK(因为只有 STACK 属性才表示该段是堆栈段),分类名选用 'STACK'。其他的逻辑段,前 3 个属性参数都选用缺省方式。

不同模块中,链接方式相同的同名段,如果有 '分类名' 的话, '分类名' 必须相同。

DOS 环境下运行的程序选用 USE16 做段长度。

3. 段约定语句

格式：ASSUME 段寄存器：段名，……，段寄存器：段名

功能：ASSUME 语句通知汇编程序，寻址逻辑段使用哪一个段寄存器。

例如：ASSUME CS: CODE,DS: DATA

这条语句通知汇编程序，以 CODE 为段名的是代码段，对代码段寻址约定使用 CS 寄存器，以 DATA 为段名的是数据段，对数据段寻址约定使用 DS 寄存器。

说明：

① ASSUME 语句是非执行语句，要求放在代码段之中，执行寻址操作之前。习惯上，把 ASSUME 语句作为代码段的第一条语句。

② ASSUME 语句，仅仅是约定了对某个逻辑段进行寻址操作时使用哪一个段寄存器，而段寄存器的初值还必须在程序中用指令设置。有两个办法可以设置段寄存器的初值，设 DATA 为数据段段名：

- 使用 SEG 运算符求出逻辑段的段基址赋给段寄存器

如：MOV AX, SEG DATA
 MOV DS, AX

- 直接把段名赋给段寄存器

如：MOV AX, DATA
 MOV DS, AX

DOS 把一个可执行程序(EXE 文件)调入内存之后，自动把程序代码段的段基址赋给了 CS,也就是说，对 CS 的赋值是由 DOS 系统自动完成的，程序员不能干预。

4. 过程定义语句

一个“过程”，通俗地说就是一个“子程序”。子程序必须用 RET 作为返回指令，用过程定义语句为子程序定界。过程定义语句的格式如下：

```
过程名      PROC 属性参数
               .....
               过程的实体
               .....
               RET
过程名      ENDP
```

过程名就是子程序的名字，它的命名规则和变量名一样。过程的属性有两个参数可供选择：

① NEAR(或者缺省)表示该过程是一个近过程。近过程的概念是说：该子程序和调用它的那条指令在同一个代码段中。

② FAR 表示远过程。远过程和调用它的那条指令不在同一个代码段。



5. 定位语句

格式: ORG 表达式

例如:

```
CODE      SEGMENT
           ASSUME    CS: CODE
           ORG       100H
BEG:      JMP        START
BUF      DB          12,34
START:    .....
           .....
CODE      ENDS
           END        BEG
```

ORG 语句通知汇编程序,从表达式给出的偏移地址开始,依次存放后续目标块,例如上例 JMP START 的目标指令应放在代码段偏移地址为 100H 的单元。

6. 汇编结束语句

汇编结束语句有两种格式。

格式 1: END 程序的启动地址标号

例如:

```
END      BEGIN
```

它通知汇编程序,源程序到此结束,用 BEGIN 作标号的指令是程序的启动指令。

DOS 装载程序的可执行文件(EXE 文件)时,自动把标号 BEGIN 所在段的段基址赋给 CS,把 BEGIN 所在单元的偏移地址赋给 IP。从而 CPU 自动从 BEGIN 开始的那条指令依次执行程序。

在单一模块的源程序中,以及在模块化程序的主模块中必须用格式 1 作为源程序的最后一条语句。

格式 2: END

END 语句通知汇编程序,源程序到此结束。在模块化程序的子模块中,必须用格式 2 作为源程序的最后一条语句。

程序在完成预定任务之后,必须返回 DOS。返回 DOS 最常用的方法是使用 DOS 系统 4CH 功能调用,即连续执行以下 3 条(或 2 条)指令:

```
MOV      AH,4CH
MOV      AL,返回码      ;如果不准备组织批处理文件,此条可省略。
INT      21H
```

习 题

1. 设数据段定义如下：

```
NUM DB +50,-1,250,87,-100,120
     DW 1234H,'A'
```

(1) 用十六进制数表示汇编后数据段单元的内容。

(2) 用十六进制写出汇编后最大的机器数,最大的真值数(写出真值),最小的真值数(写出真值)。

2. 设数据段定义如下：

```
BNUM DB +50,-1,250,87,-100,120
LLL EQU $-BNUM
DNUM DD 11223344H
WNUM DW 0FFH
```

(1) 改正下列非法指令,然后写出指令执行后 AX,BX,CL 寄存器中的值。

```
MOV AX,BNUM
MOV BX,BNUM+5
MOV CL,DNUM+2
```

(2) 访问数据段将 2233H 取出存入 AX 寄存器。

(3) 用一条指令将 BX 的值存入 DNUM 和 DNUM+1 单元。

(4) 分别执行“INC BYTE PTR WNUM”和“INC WNUM”,相关的存储单元中结果有何不同?

汇编语言程序设计

在 32 位微型计算机系统中, CPU 可以工作在实地址模式、保护模式和虚拟 8086 模式。在保护模式和虚拟 8086 模式下的程序设计较多地涉及操作系统方面的知识, 本章不做介绍。做为程序设计的基础篇章, 本章仅介绍基于 DOS 环境的实地址模式程序设计, 有关中断和接口的程序设计将分散在后续章节中介绍。程序设计必然涉及数据的输入、输出, 系统软件(DOS 和 BIOS)为此设计了许多子功能可供用户程序调用。调用系统功能也是程序设计的一项重要内容。

本章首先介绍汇编源程序的编程格式和系统功能调用, 为程序设计打下扎实的基础, 然后分类介绍程序设计的基本方法。

5.1 汇编源程序的编程格式

汇编源程序有两种编程格式: 一种格式只能生成扩展名为 EXE 的可执行文件, 称为 EXE 文件的编程格式; 另一种格式可以生成扩展名为 COM 的可执行文件, 称为 COM 文件的编程格式。COM 文件的执行级别高于 EXE 文件, 同名的 BAT(批处理)文件执行级别最低。

5.1.1 EXE 文件的编程格式

EXE 文件的编程格式允许源程序使用多个逻辑段, 在实地址模式下, 每个逻辑段的目标块不超过 64KB, 适合编写大型程序。

【例 5.1.1】 显示 10 行 HELLO。

```
                ;FILENAME: 511. ASM
                .486
DATA            SEGMENT USE16
MSG             DB          'HELLO',0DH,0AH,' $ '
DATA            ENDS
STACK_          SEGMENT PARA STACK 'STACK' USE16
                DB          100 DUP(?)
STACK_          ENDS
```

```

CODE      SEGMENT USE16
          ASSUME  CS: CODE,DS: DATA,SS: STACK_
BEG:      MOV     AX,STACK_
          MOV     SS,AX
          MOV     SP,100                ;以上置堆栈段初值(可省略)
          MOV     AX,DATA
          MOV     DS,AX                ;以上设置 DS 初值
          MOV     CX,10                ;设置循环次数
LAST:     MOV     AH,9
          MOV     DX,OFFSET MSG
          INT     21H                  ;显示一行 HELLO
          LOOP    LAST                ;循环计数
          MOV     AH,4CH
          INT     21H                  ;返回 DOS
CODE      ENDS
          END     BEG                  ;汇编结束语句

```

【结构分析】

本例源程序中有 3 个逻辑段,以 DATA 为段名的是数据段,以 STACK_为段名的是堆栈段,以 CODE 为段名的是代码段。

代码段开始用一条 ASSUME 语句设置段约定,接着给 SS、DS 寄存器赋初值,然后是一个循环程序,显示 10 行 HELLO,代码段结束用 4CH 功能返回 DOS。

源程序最后一行是汇编结束语句,它通知汇编程序源程序到此结束,程序的启动地址为 BEG。

DOS 把 EXE 文件调入内存后,会自动给 SS、SP 赋初值,因此程序中给 SS、SP 赋初值的三条指令可以省略。

5.1.2 COM 文件的编程格式

汇编语言要求 COM 文件的编程格式必须符合以下规定:

源程序只允许使用一个逻辑段,即代码段,不允许设置堆栈段;

程序使用的数据,可以集中设置在代码段的开始或末尾;

在代码段偏移地址为 100H 的单元,必须是程序的启动指令;

代码段目标块小于 64KB。

因此 COM 文件的编程格式适合于编写中小型程序。下面我们把例题 5.1.1 改造成 COM 文件的编程格式。

【例 5.1.2】 显示 10 行 HELLO。

```

          ;FILENAME: 512.ASM
          .486
CODE      SEGMENT USE16
          ASSUME  CS: CODE
          ORG     100H

```

```

BEG:      JMP      START
MSG      DB      'HELLO',0DH,0AH,'$' ;用户程序数据区
START:    MOV      CX,10                ;设置循环次数
LAST:     MOV      AH,9
          MOV      DX,OFFSET MSG
          INT      21H                  ;显示一行 HELLO
          LOOP     LAST                 ;循环计数
          MOV      AH,4CH
          INT      21H                  ;返回 DOS
CODE      ENDS
          END      BEG

```

【结构分析】

① 本程序只有一个以 CODE 为段名的代码段,用户程序使用的数据放在程序之前。用 ORG 伪指令在偏移地址为 100H 的单元放置一条 JMP START 指令用来跳过数据区。

② DOS 把 COM 文件调入内存后,将自动使 CS=DS,因此没有必要再给 DS 赋初值。

③ 在 COM 格式的文件中,返回 DOS 还可以使用 INT 20H,本例没有采用。

【COM 文件的生成】

生成 COM 文件的首要条件是源程序的编程格式必须符合上述规定。使用低版本的汇编(如 5.0 版 MASM.EXE)、链接程序(如 3.6 版 LINK.EXE),生成 COM 文件需要 4 个步骤:

- ① 编辑生成 ASM 文件;
- ② 汇编生成 OBJ 文件;
- ③ 链接生成 EXE 文件;
- ④ 用 EXE2BIN.EXE 文件将 EXE 文件转换成 COM 文件。

使用高版本的汇编(如 TASM.EXE)、链接(如 TLINK.EXE)程序只要 3 个步骤即可生成 COM 文件:

- ① 编辑生成 ASM 文件;
- ② 汇编生成 OBJ 文件;
- ③ 链接时使用小写的“t”做链接参数即可直接生成 COM 文件。

5.1.3 EXE 文件和 COM 文件的内存映像

磁盘上的 EXE 文件包括两部分:一部分为装入模块,另一部分为“重定位信息”。DOS 装载 EXE 文件时,这两部分都调入内存。DOS 测试内存环境,根据重定位信息,完成对装入模块的重定位之后,重定位信息即被丢弃。DOS 再在同一内存块的用户程序上方(低地址处)偏移地址为 00~FFH 的单元自动生成一个有 256 个字节的数据块,该数据块被称为“程序段前缀”(Program Segment Prefix,PSP)。

DOS 自动给 DS、ES、FS、GS 赋值,使 DS=ES=存放 PSP 的段基址,FS=GS=0,并

使 CS: IP=用户程序的启动地址,SS: SP 指向用户堆栈段的栈顶,在这以后,DOS 才把控制权交给用户程序。

正因为如此,所以在用户程序中(例如 511. ASM)给 SS、SP 赋初值的 3 条语句是多余的。还有一点必须注意: DOS 给 DS、ES 所赋的初值并不等于用户程序数据段、附加段的段基址,因此用户程序一开始必须对 DS(ES、FS、GS)初始化就是这个道理,EXE 文件的内存映像见图 5-1。

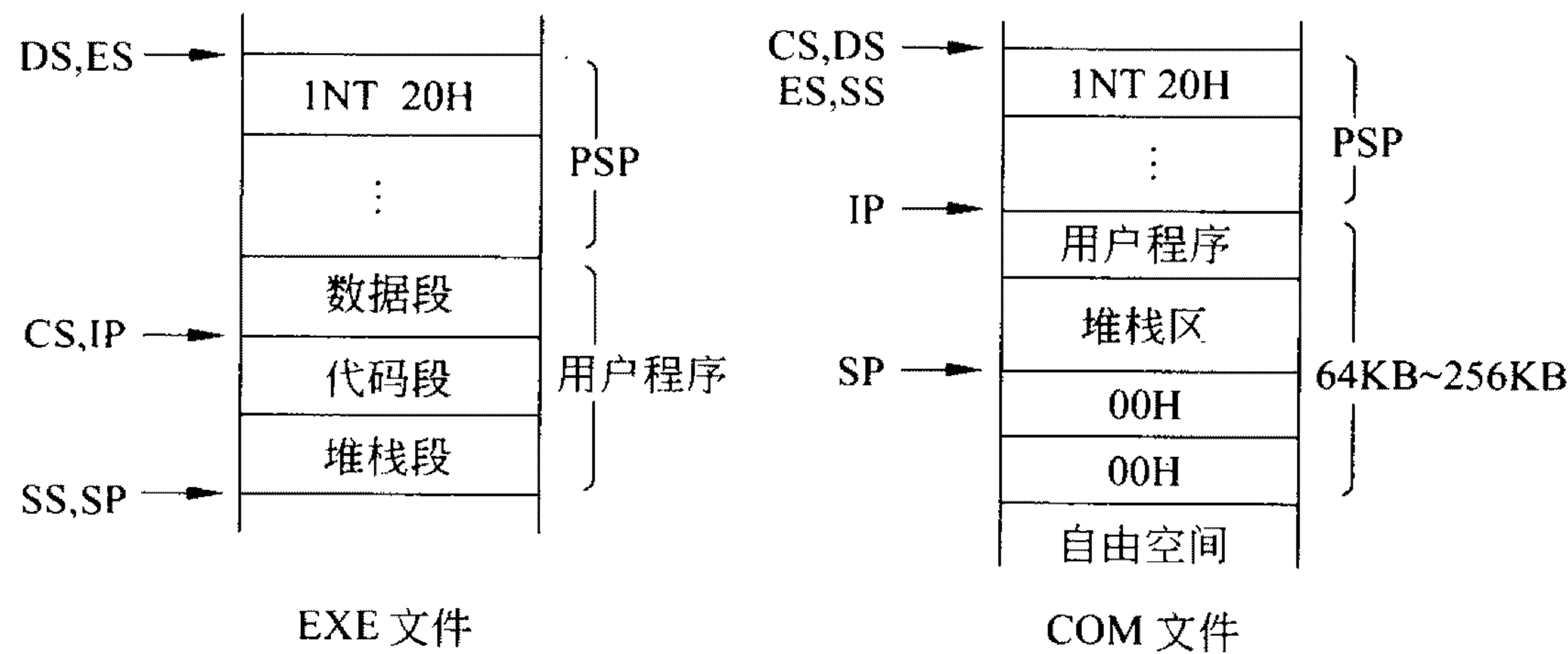


图 5-1 EXE 文件和 COM 文件内存映像

COM 文件没有重定位信息,因此比 EXE 文件的体积小得多。DOS 在装载 COM 文件时,也在用户程序上方偏移地址 00~FFH 单元生成一个程序段前缀,然后从偏移地址 100H 开始依次存放用户程序。DOS 自动赋值使 CS=DS=ES=SS=PSP 的段基址,FS=GS=0,并使 IP=100H,SP=FFFEH。随后 DOS 把控制权交给用户程序,CPU 从 CS: 100H 单元依次执行用户程序。图 5-1 形象地给出了 COM 文件的内存映像,从中看出 DOS 在装入 COM 文件之后,自动为用户程序设置了堆栈段,且堆栈段在用户程序的高端。初始栈顶为二个字节的 0,超过 64KB 的自由空间也归用户程序使用。DOS 装入 COM 文件后自动使 IP=100H,所以要求 COM 文件的汇编格式中,必须在代码段偏移地址为 100H 的单元放置程序的启动指令就是这个道理。

5.1.4 程序段前缀

程序段前缀(PSP)占用 256 个单元,其中的信息是 DOS 装载可执行文件时自动生成的。DOS 通过 PSP 向用户程序传递参数,通过 PSP 向用户程序提供程序正常结束和异常结束时返回 DOS 的途径。总之,DOS 是通过 PSP 管理用户程序的。PSP 是程序运行不可分割的部分,内存中的每一个可执行程序都伴随有该程序的程序段前缀。当程序结束返回 DOS 之后,用户程序和它的程序段前缀所占用的内存空间均被释放,交还 DOS 另行分配。程序段前缀的数据格式如下(设 PSP 为程序段前缀的段基址):

- PSP: 00~01H INT 20H
- PSP: 02~03H 可用内存的大小(以节为单位,1 节=16 字节)
- PSP: 04 保留
- PSP: 05~09H 老式的系统功能调用指令
- PSP: 0A~0DH INT 22H 中断向量

PSP: 0E~11H	INT 23H 中断向量
PSP: 12~15H	INT 24H 中断向量
PSP: 16~2BH	保留
PSP: 2C~2DH	环境变量区段基址
PSP: 2E~4FH	保留
PSP: 50~51H	INT 21H
PSP: 52H	远程 RET
PSP: 53~5BH	保留
PSP: 5C~6BH	第一个格式化的命令行参数(即未打开的 FCB1)
PSP: 6C~7BH	第二个格式化的命令行参数(即未打开的 FCB2)
PSP: 7C~7FH	保留
PSP: 80H	命令行参数的长度(不包括回车)
PSP: 81~FFH	未格式化的命令行参数

什么是命令行参数?

DOS 把可执行文件的文件名作为外部命令,在 DOS 提示符下键入文件名之后,DOS 就把相应的可执行文件调入内存执行。键入文件名的时候,如果在文件名之后空一格再键入一串字符,这串字符就称为命令行参数。

命令行参数是程序执行所需要的原始数据。使用命令行参数向程序传递数据,是程序设计的常用手段。本书第 4 章介绍的 TASM.EXE 和 TLINK.EXE 就是例证,用它们进行汇编和链接时,都必须把待汇编和待链接的文件名做为命令行参数一块键入。

未格式化的命令行参数,就是命令字之后从空格开始的原始字符串。命令行参数中不能含有 I/O 改向符“<”、“>”和管道操作符“|”,DOS 将把符合这一规定的命令行参数照原样搬移到 PSP: 81~FFH 的 127 个单元之中,供用户程序调用。

当程序使用“文件控制块”(FCB)进行磁盘文件操作时,相关的功能调用要求用户程序建立文件控制块。用户可以在用户程序中建立,也可以使用命令行参数,由用户给出待操作文件的盘符、文件名和扩展名(这些称为一个参数)。DOS 在装载可执行文件时,对命令行参数按照 FCB 的格式进行格式化,并把格式化之后的第一个参数存入 PSP: 5CH 开始的 12 个单元,格式化之后的第二个参数存入 PSP: 6CH 开始的 12 个单元,供用户程序使用。

此外,PSP: 80~FFH 这 128 个单元还有另一个用途:当用户程序使用 FCB 读写磁盘文件的时候,该区域作为系统约定的文件读写缓冲区。用户也可以使用 DOS 系统 1AH 功能调用,把文件读写缓冲区设置在用户程序数据段,而不使用 PSP: 80~FFH 这 128 个单元。

5.1.5 返回 DOS 的其他方法

对于 EXE 文件和 COM 文件,返回 DOS 最常用的方法是调用 INT 21H 的 4CH 功能。除此之外:

(1) 对于 COM 文件还有以下 3 种方法可以返回 DOS,即:

① 直接执行 INT 20H。

② 调用 INT 21H 的 0 号功能。

③ 执行 RET 指令。如果此时 SP 仍然等于 FFFEh 的话(程序中规范地使用堆栈,在执行 RET 之前把进栈的数据全部弹出,则 SP 必然等于 FFFEh),CPU 将无条件转入 PSP 的首单元,执行那里的 INT 20H,返回 DOS。

(2) 对于 EXE 文件的编程格式,还有一种返回 DOS 的方法,就是在需要返回 DOS 的时候,设法使 CPU 转到 PSP 首单元,执行那里的 INT 20H 指令。为此,源程序必须采取 3 项措施,缺一不可。

① 把整个可执行程序包含在一个远过程之中。

② 在给 DS 赋初值之前,用下列 3 条指令把 PSP 首单元的逻辑地址压入栈顶。即:

```
PUSH    DS
MOV     AX,0
PUSH    AX
```

③ 在采取了以上两条措施之后,程序在需要返回 DOS 的地方执行一条 RET 指令即可返回 DOS。因为这条 RET 指令是远过程中的返回指令,它将从栈顶弹出 4 个字节,即把 PSP 首单元的逻辑地址反弹到 CS: IP 之中,从而使 CPU 转移到 PSP 首单元执行那里的 INT 20H,再返回 DOS。

做为范例,我们把 511. ASM 改造成 513. ASM,请参阅例 5.1.3,请读者注意,例 5.1.3 的编程风格不是很好,我们不推荐。

综上所述,虽然返回 DOS 有好几种方法,但是调用 INT 21H 的 4CH 功能,是返回 DOS 最常用的方法,它的优点是简单通用,而且有利于组织批处理文件。

【例 5.1.3】 显示 10 行 HELLO。

```
                ;FILENAME: 513. ASM
                .486
DATA            SEGMENT USE16
MSG             DB      'HELLO',0DH,0AH,'$'
DATA            ENDS
STACK_          SEGMENT PARA STACK 'STACK' USE16
                DB      100 DUP(?)
STACK_          ENDS
CODE            SEGMENT USE16
                ASSUME  CS: CODE,DS: DATA,SS: STACK_
MAIN            PROC     FAR
BEG:            PUSH     DS                ;PSP 段基址压栈
                MOV     AX,0
                PUSH     AX              ;双字节 0 压栈
                MOV     AX,DATA
                MOV     DS,AX            ;DS 初始化
                MOV     CX,10           ;设置循环次数
LAST:           MOV     AH,9
                MOV     DX,OFFSET MSG
```

```

                INT      21H
                LOOP     LAST           ;循环计数
                RET              ;返回 DOS
MAIN           ENDP
CODE           ENDS
                END      BEG

```

5.1.6 源程序堆栈段的设置

汇编源程序有两种编程格式,按 COM 格式编写的源程序不允许设置堆栈段,COM 文件装入内存后,DOS 将自动在用户程序高地址处为其设置堆栈段。

按 EXE 格式编写的源程序,如何设置堆栈段呢? 回答是: 可以设置用户堆栈段,也可以不设置。前文已经介绍过,如果不设置堆栈段,链接时链接程序将给出错误信息: “Warning no stack segment”。DOS 把这种 EXE 文件装入内存后,自动为其分配不少于 128 个字节的堆栈段。用户程序执行“CALL”指令,执行“INT n”指令,或者响应中断的时候,CPU 将自动把断口地址压入堆栈。在实地址模式下,一个断口地址只有 2~6 个字节,如果用户程序不是多层次地嵌套调用子程序,而且用户程序执行“PUSH”指令压入堆栈的数据不是很多的话,DOS 分配的这 128 个字节的堆栈段是绰绰有余的。鉴于此,本书在后续章节的例题中,都没有设置堆栈段。

5.2 DOS 系统 I/O 功能调用

DOS 有 4 个组成部分,其中 IBMBIO.COM 和 IBMDOS.COM 是 DOS 系统的核心模块。IBMBIO.COM 是基本 I/O 设备处理程序,它完成数据输入和数据输出的基本操作,而 IBMDOS.COM 是磁盘文件管理程序,这两个模块均有若干子功能可以被用户程序调用。用户程序调用这些子功能,称之为“DOS 系统功能调用”。

用户程序通过 INT 21H 软中断指令调用 DOS 系统功能,调用模式如下:

```

MOV    AH,功能号
设置入口参数
INT    21H
分析出口参数

```

每个子功能都有一个功能号。调用时,需要把功能号做为立即数赋给 AH 寄存器。大部分子功能都有一定的运行条件,调用时要根据要求,设置相关的入口参数,然后执行 INT 21H 指令,转入系统的“21H”型中断服务程序,执行其中的由 AH 指定的子程序。子程序执行完毕,自动返回,执行 INT 21H 的后续指令。有些子程序执行后,带回执行结果,因此返回后,用户程序通常要获取或者分析出口参数。

本小节仅介绍 DOS 系统常用的输入/输出功能调用。

【功能号 00H】结束一个程序。

入口参数: CS=程序段前缀段基址。

在 COM 格式的源程序中,运用此项功能,可以结束一个程序,返回 DOS。

【功能号 01H】等待键入一个字符,有回显,响应 Ctrl_C。

入口参数:无。

出口参数:AL=按键的 ASCII 码。若 AL=0,表明按键是功能键或光标键,必须再次调用本功能,才能返回按键的扩展码。

【功能号 02H】显示一个字符,响应 Ctrl_C。

入口参数:DL=待显字符的 ASCII 码。

出口参数:无。

本功能在屏幕的当前位置显示一个字符,光标右移一格,如果是在一行末尾显示字符,则光标返回下一行的开始格。如果是在屏幕的右下角显示字符,光标返回时屏幕要上滚一行。实验表明,该项功能要破坏 AL 寄存器的内容。

【功能号 03H】从主串口读一个字符。

入口参数:无。

出口参数:AL=从主串口读到的字符编码。

【功能号 04H】向主串口写一个字符。

入口参数:DL=欲输出的字符编码。

出口参数:无。

【功能号 05H】向打印机发送一个字符。

入口参数:DL=待打印字符的 ASCII 码。

出口参数:无。

该项功能,DOS 将自动检测打印机,如果打印机未准备好,缺纸、打印缓冲区满,DOS 将在屏幕上显示错误信息。

【功能号 06H】字符显示/字符输入。

该项功能根据 DL 寄存器的内容执行字符显示/字符输入功能。若 DL=FFH,该项功能为字符输入,若 DL 不等于 FFH 该项功能为字符显示。

入口参数:DL=0~FEH。

出口参数:无,显示与 DL 内容对应的字符。

入口参数:DL=FFH。

出口参数:该项功能执行时,若键盘缓冲区无字符可取,则置 Z 标志为 1。若有字符可取,则置 Z 标志为 0,AL 寄存器即为输入字符的 ASCII 码。如果 AL=0,必须再次调用该项功能,方能在 AL 中得到按键的扩展码。做为字符输入功能时,不回显,不响应 Ctrl_C。

【功能号 07H】等待键入一个字符,无回显,不响应 Ctrl_C。

入口参数:无。

出口参数:AL=按键的 ASCII 码,若 AL=0,需再次调用该项功能才能在 AL 中得到按键的扩展码。

【功能号 08H】等待键入一个字符,无回显,响应 Ctrl_C。

入口参数：无。

出口参数：AL=按键的 ASCII 码,若 AL=0,需再次调用该项功能才能在 AL 中得到按键的扩展码。

【功能号 09H】显示字符串,响应 Ctrl_C。

入口参数：DS: DX=字符串首地址,字符串必须以 '\$'(即 ASCII 码 24H)为结束标志。

出口参数：无。

该项功能从屏幕当前位置开始,显示字符串,遇到结束标志 '\$' 时停止, '\$' 字符并不显示。实验证明 9 号功能也破坏 AL 寄存器的内容。

【功能号 0AH】等待键入一串字符送入用户程序的数据缓冲区。

入口参数、出口参数如图 5-2 所示。

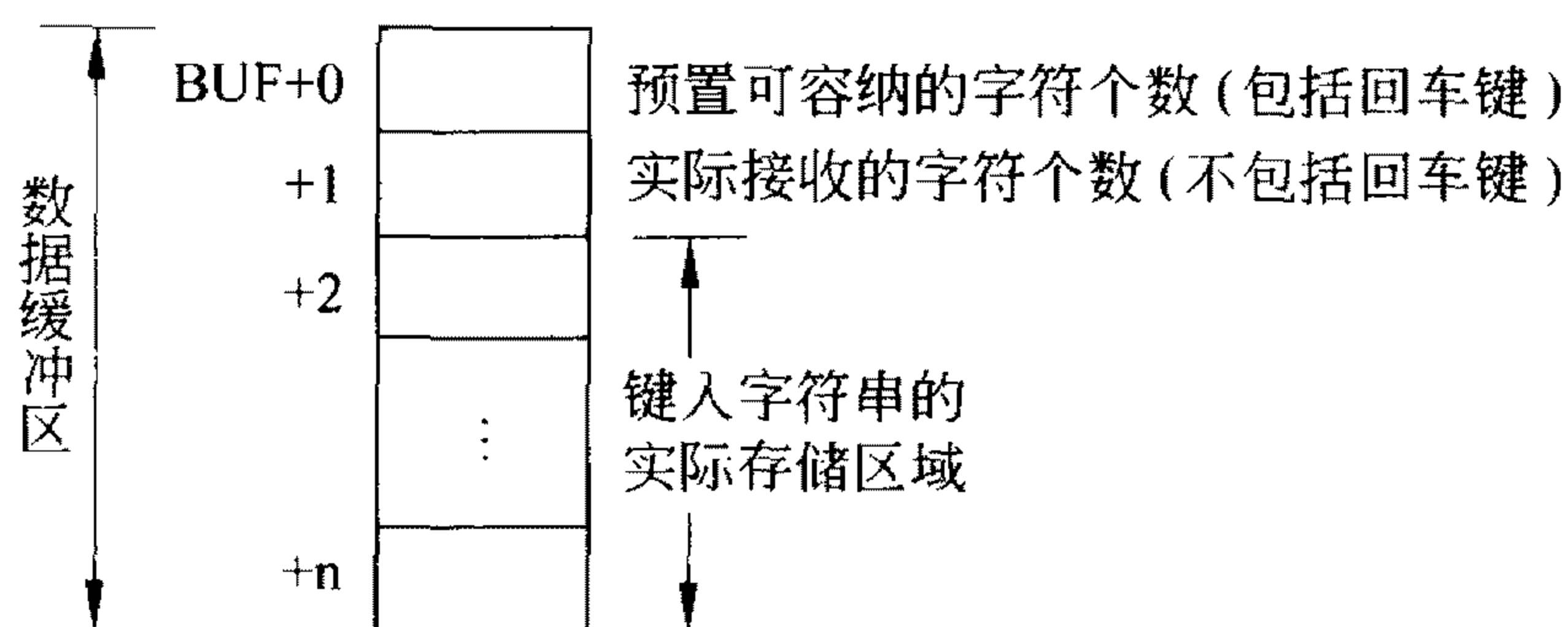


图 5-2 0AH 功能要求的数据缓冲区格式

① 0AH 功能要求键入的字符串以“回车”做为结束标志,换句话说,按下回车键之后,本次功能调用结束,光标返回当前行始格。“回车”符留在缓冲区当中。

② 对数据缓冲区的格式有如下要求:

缓冲区要设置在用户程序数据段,调用前,缓冲区首址偏移量应赋给 DX 寄存器。

缓冲区首单元应预置“允许接收的字符个数”(包括回车键在内)。

用户键入回车之后,由 0AH 功能把实际键入的字符个数(不包括回车键)写入 BUF+1 单元。

键入的字符串从 BUF+2 单元开始依次存放。因此缓冲区的容量要大于(或等于)键入串的长度(包括回车键)+2。

缓冲区不接收超长字符,并发出声响以示警告。

③ 0AH 功能在接收字符的过程中,有回显,响应 Ctrl_C,按下退格键可删除屏幕以及缓冲区的当前字符。

下面给出了 0AH 功能调用的示范,它允许用户键入 15 个字符(包括回车键)。

假设数据段	BUF	DB	15
		DB	?
		DB	15 DUP(?)
代码段	
		MOV	AH,0AH
		MOV	DX,OFFSET BUF
		INT	21H
	

【功能号 0BH】查询有无键盘输入,响应 Ctrl_C。

入口参数:无。

出口参数:AL=0 无键入。

AL=FFH 有键入。

【功能号 0CH】清除键盘缓冲区,然后调用由 AL 指定的功能。

入口参数:

① AL 允许是:01H 键入一个字符,有回显,响应 Ctrl_C。

06H 字符 I/O。

07H 键入一个字符,无回显,不响应 Ctrl_C。

08H 键入一个字符,无回显,响应 Ctrl_C。

0AH 键入一串字符,写入用户程序数据区。

② 其他入口参数应根据相关的功能,另行设置。

【功能号 4CH】

该项功能,终止当前程序的运行,并把控制转交给调用它的程序。由被终止程序打开的全部文件都被关闭。该项功能还把程序占用的内存空间交还 DOS 另行分配。

该项功能,允许被终止的程序传送一个“返回码”给调用它的程序。假若被终止的程序是由 DOS 命令调入的,那么返回码可以被 DOS 批处理命令中的 IF ERRORLEVEL 子命令识别。

入口参数:AL=返回码(或者不设置)。

出口参数:无。

【说明】

当用户程序执行 1、2、8、9、0AH、0BH、0CH 功能调用时,如果键入 Ctrl_C 或者 Ctrl_Break,DOS 将自动调用 INT 23H 中断处理程序,从而强行使用户程序中断,称为响应 Ctrl_C。

下面通过一个例题,学习 DOS 功能的调用方法。

【例 5.2.1】人机会话

通过人机会话可以获取程序运行所需要的参数。本例要求程序运行后,询问用户姓名并等待回答,用户输入姓名后按下回车键,程序再把键入的姓名复制在屏幕上等待用户认可,用户键入“Y”程序结束,否则再次询问用户姓名。

【设计思路】

设计人机会话程序,首先要确定屏显格式,本例假设屏显格式如下:

What is your name? Li nei	;前者为程序输出的询问信息,后者是用户键入的回答
Li nei ? (Y/N) N	;前者是复制的姓名和回答提示,后者是用户键入的 N
What is your name? Li mei	;再次问答
Li mei ? (Y/N) Y	;再次复制的姓名与用户回答

本例做为 DOS 功能调用的示范,涉及 1、2、9、0AH 4 个功能调用。



【程序清单】

```

;FILENAME: 521. ASM
.486
DATA SEGMENT USE16
MSG1 DB 0DH,0AH,'What is your name ? $ '
MSG2 DB '? (Y/N) $ '
BUF DB 30
DB ?
DB 30 DUP(?)
DATA ENDS
CODE SEGMENT USE16
ASSUME CS: CODE,DS: DATA
BEG: MOV AX,DATA
MOV DS,AX
AGAIN: MOV AH,9
MOV DX,OFFSET MSG1
INT 21H ;询问姓名
MOV AH,0AH
MOV DX,OFFSET BUF
INT 21H ;接收用户键入的字符串
MOV BL,BUF+1
MOV BH,0 ;实际键入的字符个数→BX
MOV SI,OFFSET BUF+2
MOV BYTE PTR [BX+SI],'$ ' ;用'$ '做为串结束符
MOV AH,2
MOV DL,0AH
INT 21H ;使光标下移一行
MOV AH,9
MOV DX,OFFSET BUF+2
INT 21H ;复制用户键入的字符串
MOV AH,9
MOV DX,OFFSET MSG2
INT 21H ;给出认可信息
MOV AH,1
INT 21H ;接收键入的一个字符
CMP AL,'Y' ;比较
JNE AGAIN
MOV AH,4CH
INT 21H ;返回 DOS
CODE ENDS
END BEG

```

5.3 BIOS 键盘输入功能调用

主机板的只读存储器(ROM)中,装有基本输入/输出系统程序(Basic Input/Output System, BIOS)。BIOS 提供了系统加电自检,引导装入,主要输入/输出设备的驱动程序,以及对系统接口电路的初始化编程等。

DOS 和 BIOS 是两组系统软件,和 DOS 功能相比,BIOS 功能更加接近系统硬件,是最底层的系统软件,而 DOS 功能则是较高层次的系统软件,DOS 的许多功能是调用 BIOS 实现的。

用户程序也可以调用 BIOS 功能,我们称为“BIOS 功能调用”,调用 BIOS 功能,可以使程序获得较高的运行效率。BIOS 功能调用模式如下:

```
MOV    AH,功能号
设置入口参数
INT     n
分析出口参数
```

其中 INT n 为软中断指令,n 为中断类型码。调用 BIOS 键盘输入子功能,使用 INT 16H。调用 BIOS 屏幕操作功能,使用 INT 10H。

BIOS 常用的键盘输入功能调用如下:(调用指令 INT 16H)

【功能号 00H】读取键入的一个字符,无回显,响应 Ctrl_C,无键入则等待。

入口参数:无。

出口参数:AL=键入字符的 ASCII 码。若 AL=0,则 AH=输入键的扩展码。

【功能号 01H】查询键盘缓冲区。

入口参数:无。

出口参数:

① Z 标志=0,表示有键入,此时 AL=键入字符的 ASCII 码,AH=键入字符的扩展码。注意:该功能调用结束后,键代码仍留在键盘缓冲区中。

② Z 标志=1,表示无键入。

【功能号 02H】读取当前转换键状态。

入口参数:无。

出口参数:AL=键盘状态字。

AL₇ 位置 1 表示 Insert 键有效(被奇数次按下)。

AL₆ 位置 1 表示 Caps Lock 键有效(相应的指示灯亮)。

AL₅ 位置 1 表示 Num Lock 键有效(相应的指示灯亮)。

AL₄ 位置 1 表示 Scroll Lock 键有效(相应的指示灯亮)。

AL₃ 位置 1 表示按下了 Alt 键。

AL₂ 位置 1 表示按下了 Ctrl 键。

AL₁ 位置 1 表示按下了左 Shift 键。

AL₀ 位置 1 表示按下了右 Shift 键。

【功能号 10H】读扩展键盘,无回显,响应 Ctrl_C。

入口参数:无。

出口参数:AL=键入字符的 ASCII 码,若 AL=0,则 AH=键入字符的扩展码。

【功能号 11H】查询扩展键盘缓冲区。

入口参数:无。

出口参数:

① Z 标志=0,表示有键入,此时 AL=键入字符的 ASCII 码,AH=键入字符的扩展码。注意:该功能调用结束后,键代码仍留在键盘缓冲区中。

② Z 标志=1,表示无键入。

【功能号 12H】读取扩展键盘的转换键状态。

入口参数:无。

出口参数:AL=扩展键盘状态字,AL₇~AL₀ 的置位条件同功能号 02H。

5.4 文本方式 BIOS 屏幕功能调用

BIOS 系统软件中,有些子功能是对屏幕进行操作的,在介绍这类功能之前必须了解一些显示器的基本知识。

5.4.1 显示器

CRT(Cathode Ray Tube,阴极射线管)显示器是微型计算机系统的输出设备。PC 系列机可以配置单色显示器,或彩色显示器。

显示适配器是显示器和系统总线之间的接口电路。适配器插在主机箱的扩展槽中通过 9 芯插座和 CRT 显示器相连。显示适配器有很强的功能,本身有 CRT 控制器、定时器、字符发生器、显示存储器等。经过初始化编程,能够独立地提供显示器工作所需的各种信号。显示器和显示适配器共同组成微型计算机的输出显示系统。

1. 屏显方式

在单色适配器支持下,单色显示器只能工作在文本方式,显示字母、数字、符号。在彩色图形适配器支持下,彩色显示器可以工作在文本方式,也可以工作在图形方式。工作在文本方式,屏幕可以显示 40 列×25 行字符,或者 80 列×25 行字符。启动 DOS 以后,适配器自动工作在 80 列×25 行的文本方式。

2. 显示存储区

系统 RAM 部分空间被指定作为显示存储区。当适配器工作在 80 列×25 行文本方式时,屏幕被划分为 80 列×25 行,共有 2000 个“方格”,每一方格显示一个字符。屏幕上每一方格对应着显示存储区(又称视频映像区)中的两个单元。显示存储区偶地址单元,

存放待显字符的 ASCII 码,紧接着的后续单元,即奇地址单元存放待显字符的“属性字”。

单显适配器中,显示存储区容量为 4KB,实际使用 4000 个单元,存放一屏字符的信息(2000 个字符的 ASCII 码,外加它们的属性字)。单显存储区的地址为 B000: 0000~0FFFH。

高档微型计算机彩色/图形适配器中,文本显示存储区容量为 32KB,分成 8 页,页号依次为 0~7,每一页存放一屏字符信息,第 0 页存储区的地址为 B800: 0000~0FFFH,第 1 页存储区的地址为 B900: 0000~0FFFH,...,第 7 页存储区的地址为 BF00: 0000~0FFFH,每一页都空闲 96 个单元。启动 DOS 后,系统默认第 0 页是“当前页”。

图 5-3 针对 80×25 文本方式形象地描述了显示存储区与屏幕字符的对应关系。字符在屏幕上的行、列号和它的 ASCII 码在显示存储区中的位置(即偏移地址),有如下关系式:

本页显示存储单元的偏移地址 = 行号 × 160 + 列号 × 2

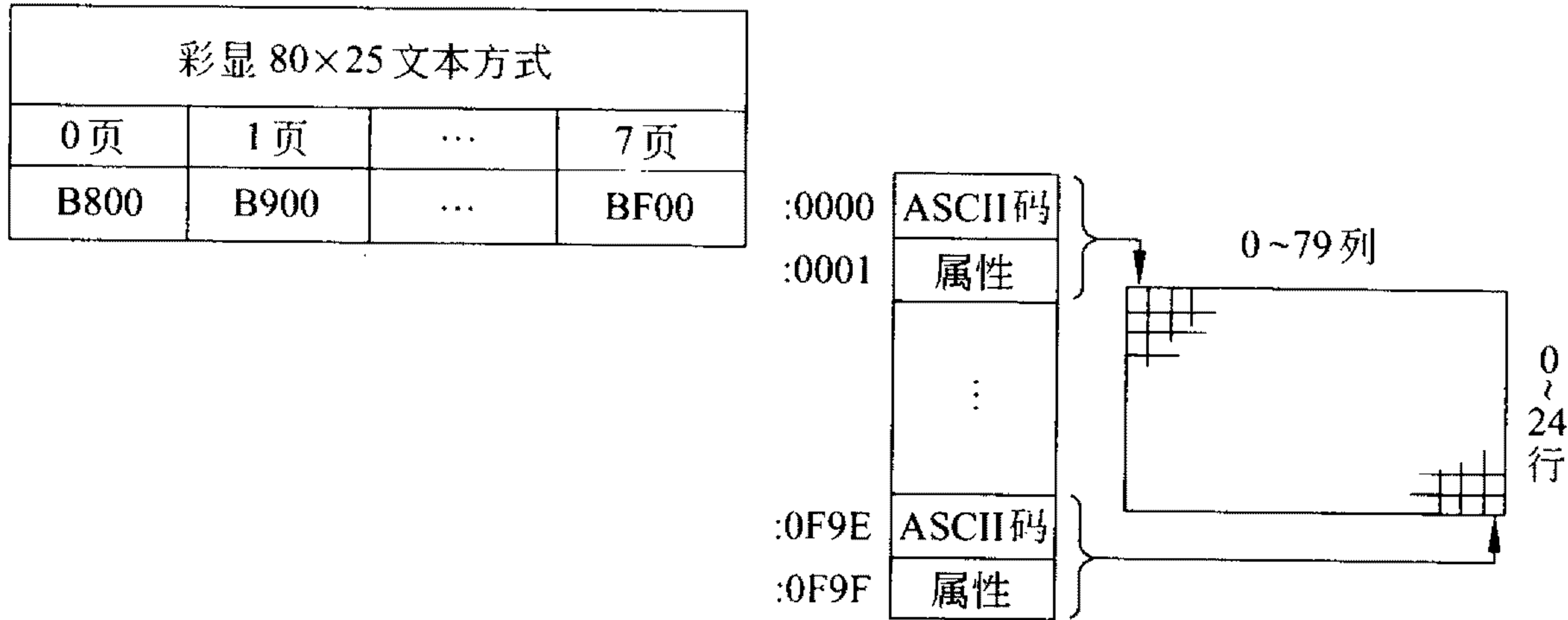


图 5-3 显示存储区与屏幕字符的对应关系

3. 属性字

属性字描述了字符显示的特性,分为单显属性字和彩显属性字两种。属性字格式如图 5-4 所示。若屏幕工作在黑白文本方式,不同的单显属性字可以显示黑底白字,白底黑字,也可以使字符闪烁显示或加亮显示。若屏幕工作在彩色文本方式,不同的彩显属性字可以使字符呈现不同的颜色,如表 5-1 所示。

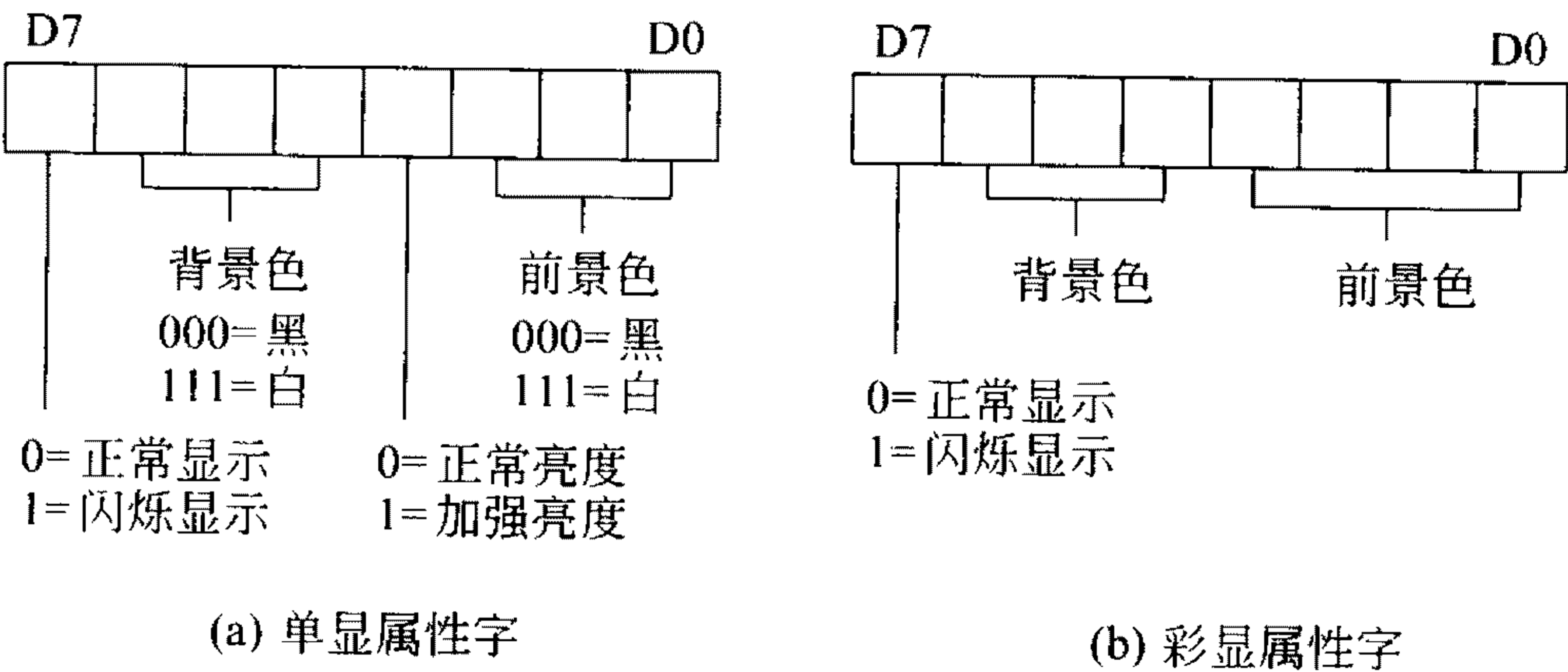


图 5-4 文本方式属性字格式

表 5-1 彩色编码表

D7 D6 D5 D4(背景) D3 D2 D1 D0(前景)	颜 色	D3 D2 D1 D0(前景)	颜 色
0 0 0 0	黑	1 0 0 0	灰
0 0 0 1	蓝	1 0 0 1	浅蓝
0 0 1 0	绿	1 0 1 0	浅绿
0 0 1 1	青	1 0 1 1	浅青
0 1 0 0	红	1 1 0 0	浅红
0 1 0 1	品红	1 1 0 1	浅品红
0 1 1 0	棕	1 1 1 0	黄
0 1 1 1	灰白	1 1 1 1	白

4. 屏幕显示

系统加电后, BIOS 对显示适配器进行初始化编程, 使屏幕工作在 80×25 黑白文本方式。在此以后, CRT 控制器就从第 0 页的显示存储区单元依次取出字符的 ASCII 码, 再转换成字符的点阵码送到 CRT, 驱动 CRT 显示。CRT 控制器以每秒钟超过 50 屏的速率, 周期性地不间断地进行上述操作。因此能够收到字符稳定显示的效果。

从程序员的角度来讲, 只要把待显字符的 ASCII 码和它的属性字写入显示存储区的相关单元, 就相当于输出到屏幕上了。显然, 对显示存储区(即视频映像区)直接进行读写操作, 速度是最快的。

BIOS 提供了一组屏幕显示的子功能, 用户程序通过 INT 10H 软中断指令可以调用它们。在通常情况下, 用户程序并不直接读写显示存储区。

5.4.2 文本方式 BIOS 屏显功能调用

调用指令 INT 10H。

【功能号 00H】设置屏幕显示方式, 兼有清屏功能。

入口参数: AL=0 40×25 黑白文本方式。
AL=1 40×25 彩色文本方式。
AL=2 80×25 黑白文本方式。
AL=3 80×25 彩色文本方式。

出口参数: 无。

【功能号 01H】设置光标形状。

入口参数: CH=光标顶部扫描线行号(0~7), CL=光标底部扫描线行号(0~7)。

出口参数: 无。

说明: 在文本方式中, 显示的每一个字符都是由若干个细小的点组成的, 称为点阵字符, 光标也不例外。彩色图形适配器生成的每一个字符由 8 行点组成, 顶部扫描行为第 0

行,低部扫描行为第7行。在这里,“行”的概念是一个点阵字符的扫描行,而不是字符的显示行(一屏有25个显示行)。

对于彩色/图形适配器,默认值为 $CH=6, CL=7$,所以光标的形状是一条闪烁的下划线,下划线的粗细占用两个扫描行的位置。如果置 $CH=3, CL=7$,光标就是一个闪烁的小方块。置 $CH=20H$,光标将消失。

【功能号 02H】 预置光标位置。

入口参数: BH =显示页号, DH =行号, DL =列号。

出口参数: 无。

【功能号 03H】 读取光标的当前位置。

入口参数: BH =显示页号。

出口参数: CH, CL =光标顶部扫描线、低部扫描线的行号。

DH, DL =光标在屏幕上的行、列号。

【功能号 05H】 设置当前显示页。

入口参数: AL =显示存储器页号 0~7。

出口参数: 在屏幕上显示出指定显示页的字符(只对文本方式有效)。

【功能号 06H】 窗口上滚。

入口参数: AL =窗口上滚的行数, BH =低部空白行的属性字。

CH, CL =窗口左上角的行、列号, DH, DL =窗口右下角的行、列号。

出口参数: 无。

说明:

① 该项功能允许在屏幕上定义一个窗口,并使窗口信息上滚 AL 行。

② 窗口是屏幕上一个局部的显示框,窗口上滚一行,即:窗口顶部行信息移出窗口之外而消失,窗口低部自动留出一行空白行,低部空白行的属性由 BH 中的属性字决定。

③ 若 $AL=0$,则窗口信息全部移出。

【功能号 07H】 窗口下滚。

入口参数: AL =窗口下滚的行数, BH =顶部空白行属性字。

CH, CL =窗口左上角的行、列号, DH, DL =窗口右下角的行、列号。

出口参数: 无。

【功能号 08H】 读取光标所在位置的字符及其属性。

入口参数: BH =显示页号。

出口参数: AH =光标所在位置的字符属性。

AL =光标所在位置的字符的 ASCII 码,如果没有对应于字符的 ASCII 码则 AL 置 0。

【功能号 09H】 从光标所在位置开始,显示若干个相同的字符。

入口参数: AL =待显示字符的 ASCII 码, BH =显示页号。

BL =待显示字符的属性, CX =重复显示的字符数。

出口参数: 无。

【功能号 0AH】 从光标所在位置开始,显示若干个相同的字符。

入口参数: AL=待显示字符的 ASCII 码, BH=显示页号,
CX=重复显示的字符个数。

出口参数: 无。

说明: 09H 和 0AH 功能均不改变光标的位置, 字符显示满一行后, 有自动换行的功能。但 09H 功能可以设置字符属性, 而 0AH 功能不能设置字符属性。

【功能号 0EH】显示一个字符。

入口参数: AL=待显示字符的 ASCII 码。

出口参数: 无。

说明: 该项功能在光标的当前位置显示一个字符, 随后光标前进一格, 满一行后自动换行, 满一页则整屏信息上滚一行。该功能和 DOS 系统的 2 号功能调用具有相同的效果。

【功能号 13H】显示字符串。

入口参数: AL=0~3, BH=显示页号, BL=属性字(当 AL=0、1 时有效), CX=串长度, DH、DL=字符串显示的起始行号、列号, ES: BP=待显示字符串首地址。

出口参数: 无。

说明:

- ① 该功能从屏幕的指定位置开始显示一串彩色字符。
- ② 待显字符串需放在附加段, 首地址偏移量需写入 BP 寄存器。
- ③ AL=0 表示: 待显示字符串中仅包含字符的 ASCII 码, 串中各字符的属性由 BL 中的属性字决定, 串显示结束后, 光标返回到调用前的位置。
AL=1 表示: 待显示字符串中仅包含字符的 ASCII 码, 串中各字符的属性由 BL 中的属性字决定, 串显示结束后, 光标停留在字符串的末尾。
AL=2 表示: 待显示字符串中包含有各个字符的 ASCII 码和属性字, 格式为 ASCII 码, 属性, ……, ASCII 码, 属性。串显示结束后, 光标返回到调用前的位置。
AL=3 表示: 待显字符串中包含有各个字符的 ASCII 码和属性字, 格式同上。串显示结束后, 光标停留在字符串的末尾。
- ④ 当 AL 选择 2 或 3 的时候, CX 中的串长度不包括各字符的属性字。

5.5 分支程序

分支程序有 3 种结构, 即: 简单分支、复合分支和多分支。

1. 简单分支

通常是在执行了算术比较指令 CMP, 或者逻辑比较指令 TEST 之后, 根据 Z、S、O、P、C 各种标志的状态进行有条件转移。

【例 5.5.1】二进制数显示程序。

将 BX 寄存器的内容以二进制数格式显示在屏幕上。

【程序清单】

```

;FILENAME: 551. ASM
.486
CODE SEGMENT USE16
    ASSUME CS: CODE
BEG:  MOV     BX,5678H
      MOV     CX,16
LAST: MOV     DL,'0'
      RCL     BX,1           ;BX 循环左移一位
      JNC     NEXT          ;进位标志为 0 转,显示 0
      MOV     DL,'1'        ;否则显示 1
NEXT: MOV     AH,2
      INT     21H
      LOOP    LAST
      MOV     AH,4CH
      INT     21H
CODE  ENDS
      END     BEG

```

该程序依次把 BX 寄存器左移一位,判断进位标志,若进位标志为 0,屏幕上显示 0;若进位标志为 1,屏幕上显示 1。

2. 复合分支

【例 5.5.2】 复合判断。

设 NUMBER 单元的数 X 以及数值 N1、N2 均为单字节无符号数,请判断 X 的大小,并根据判断结果分别显示: $N1 \leq X \leq N2$, 或 $X < N1$, 或 $X > N2$ 。

【程序清单】

```

;FILENAME: 552. ASM
.486
DATA SEGMENT USE16
MSG1 DB      'N1<= X <= N2 $'
MSG2 DB      'X < N1 $'
MSG3 DB      'X > N2 $'
NUMBER DB     ?           ;无符号数 X
N1 EQU      22
N2 EQU      88
DATA ENDS
CODE SEGMENT USE16
    ASSUME CS: CODE,DS: DATA
BEG:  MOV     AX,DATA
      MOV     DS,AX
      MOV     DX,OFFSET MSG1

```

```

                CMP     NUMBER,N1
                JNC     NEXT           ;X≥ N1 转移
                MOV     DX,OFFSET MSG2
                JMP     DISP
NEXT:           CMP     NUMBER,N2+1
                JC      DISP           ;X≤ N2 转移
                MOV     DX,OFFSET MSG3
DISP:           MOV     AH,9
                INT     21H           ;显示结果信息
                MOV     AH,4CH
                INT     21H           ;返回 DOS
CODE           ENDS
                END      BEG

```

3. 多分支

多分支结构相当于一个多路开关,在程序设计中通常是根椐某寄存器或某单元的内容进行程序转移。

在设计多分支转移程序时,如果分支太多,则平均转移速度太慢。例 5.5.3 采用转移地址表实现多分支转移,可以提高平均转移速度。

【例 5.5.3】 多分支转移。

设计一个 256 分支的段内转移程序:

设: JUMP 单元有一个数 X,若 X=0 转移到标号为 P000 的程序段。

X=1 转移到标号为 P001 的程序段。

... ..

X=255 转移到标号为 P255 的程序段。

【程序清单】

```

                ;FILENAME: 553. ASM
                . 486
DATA           SEGMENT USE16
JUMP           DB      ?           ;某数 X
TAB            DW      P000        ;汇编程序自动把标号
                DW      P001        ;P000~P255 的偏移地址
                ... ..           ;写入相应单元
                DW      P255
DATA           ENDS
CODE           SEGMENT USE16
                ASSUME  CS: CODE,DS: DATA
BEG:           MOV     AX,DATA
                MOV     DS,AX
                MOV     BL,JUMP
                MOV     BH,0

```

```

P000:      ... ..
P001:      ... ..
           ...
P255:      ... ..
CODE      ENDS
           END

```

5.6 循环程序

循环程序的结构分为单循环、双循环和多重循环。三重以上的循环程序就比较复杂了。从结构上讲,循环程序分为循环准备、循环体和循环控制三部分,通常使用寄存器或者内存单元做为循环计数器。循环程序的结构如图 5-5 所示。

【例 5.6.1】 找最大数。

假设从 BUF 单元开始是一个 ASCII 码字符串,找出其中的最大数并且送屏幕显示。程序框图见图 5-6。

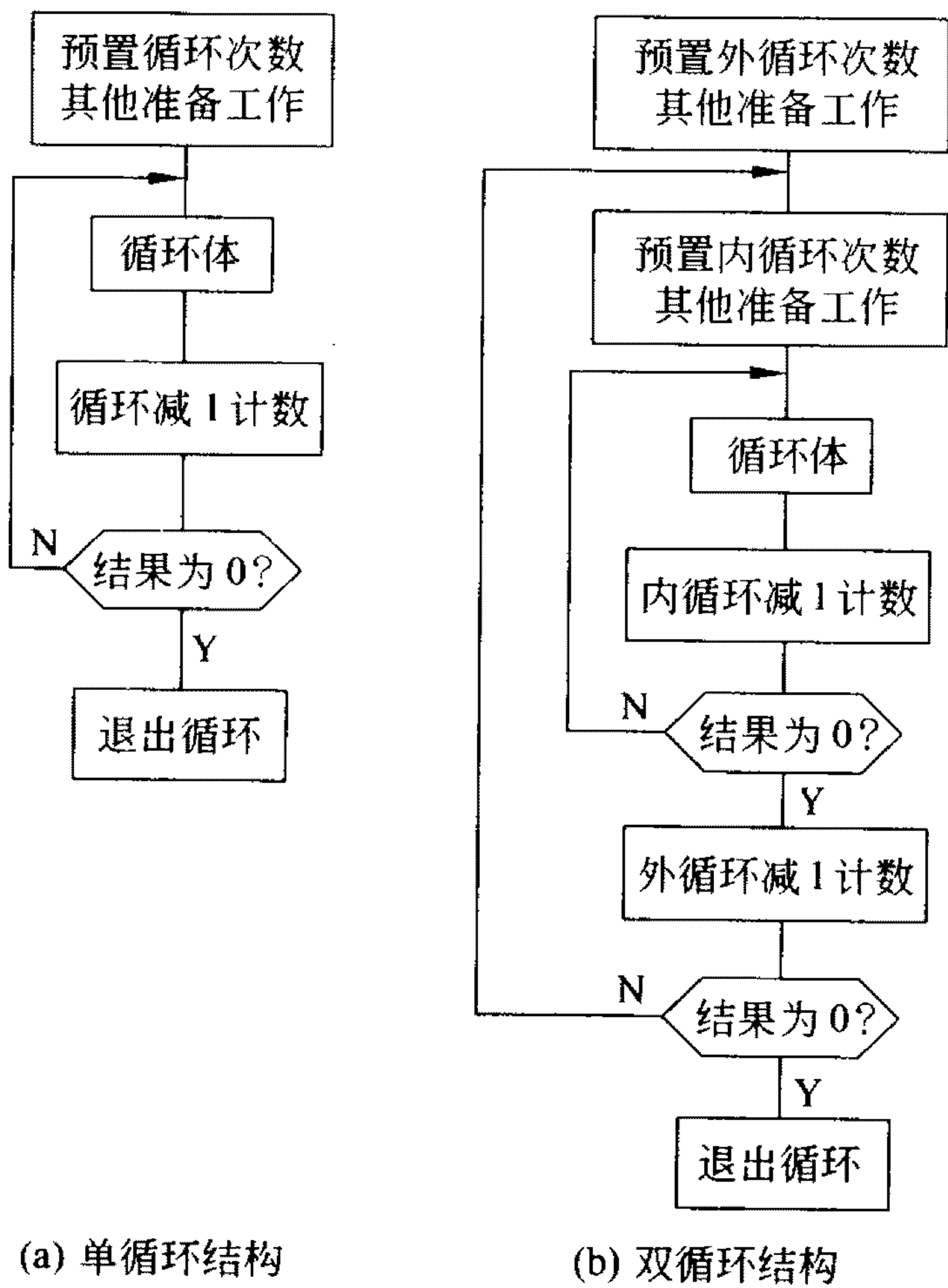


图 5-5 循环程序的结构

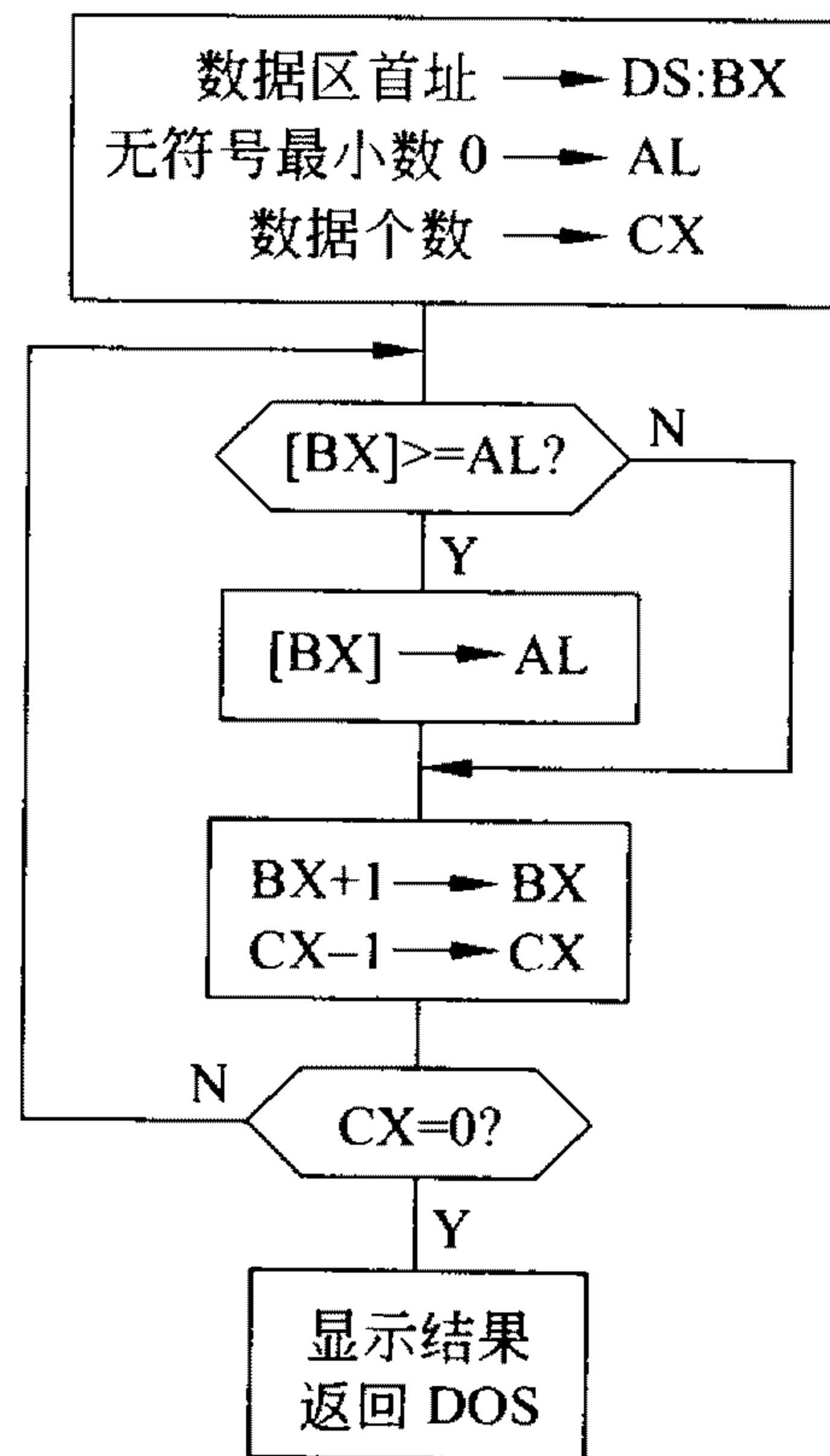


图 5-6 找最大数程序框图

【解法 1 程序清单】

```
;FILENAME: 561 1. ASM
```



```

.486
DATA    SEGMENT USE16
BUF      DB      'QWERTYUIOP123'
COUNT  EQU      $-BUF          ;统计串长度
MAX      DB      'MAX=','?',0DH,0AH,'$'
DATA     ENDS
CODE     SEGMENT USE16
        ASSUME    CS: CODE,DS: DATA
BEG:     MOV      AX,DATA
        MOV      DS,AX
        MOV      AL,0           ;无符号最小数 0 → AL
        LEA      BX,BUF        ;串首址偏移量→BX
        MOV      CX,COUNT      ;串长度→CX
LAST:    CMP      [BX],AL       ;比较
        JC       NEXT
        MOV      AL,[BX]       ;大数→AL
NEXT:    INC      BX
        LOOP     LAST          ;循环计数
        MOV      MAX+4,AL      ;最大数→MAX+4 单元
        MOV      AH,9
        MOV      DX,OFFSET MAX
        INT      21H           ;显示结果
        MOV      AH,4CH
        INT      21H           ;返回 DOS
CODE     ENDS
        END      BEG

```

ASCII 码字符应看做无符号数,而无符号数的最小值为 0,所以第一次比较的时候,把 0→AL 寄存器,各个数都和 AL 相比,每次比较都把较大的数放入 AL 寄存器中,N 个数需要比较 N 次,如果把第一个数送 AL 做为初始比较对象,那么 N 个数只需要比较 N-1 次。

在循环程序中,循环计数控制着循环体的执行次数,通常使用寄存器或者内存单元做循环计数器,也可以使用“循环结束标志”来控制循环。我们仍以找最大数为例,在下面的解法 2 程序中,在 BUF 单元字符串的末尾,额外增设了一个 FLAG 单元,其中存放“-1”,它就是字符串的结束标志。程序首先判断 BX 间址的单元内容是否是一1,若是,表明在此之前的数都判断过了,可以结束循环了。什么数可以做循环结束标志呢?不属于要比较的数据都可以做循环结束标志。采用循环结束标志来结束循环,可以不使用循环计数器,在多重循环的程序中可以省去一重循环计数。

【解法 2 · 程序清单】

```

;FILENAME: 561_2. ASM
.486
DATA    SEGMENT USE16

```

```

BUF      DB      'QWERTYUIOP123'
FLAG     DB      -1                      ;设置串结束标志
MAX      DB      'MAX=','?',0DH,0AH,'$'
DATA     ENDS
CODE     SEGMENT USE16
        ASSUME  CS: CODE,DS: DATA
BEG:     MOV     AX,DATA
        MOV     DS,AX
        MOV     AL,0                    ;无符号最小数 0 → AL
        LEA     BX,BUF                  ;串首址偏移量 → BX
LAST:    CMP     BYTE PTR [BX],-1       ;[BX]=串结束标志 ?
        JE      DISP                    ;是,转
        CMP     [BX],AL
        JC      NEXT
        MOV     AL,[BX]                  ;大数 → AL
NEXT:    INC     BX
        JMP     LAST
DISP:    MOV     MAX+4,AL                 ;最大数 → MAX+4 单元
        MOV     AH,9
        MOV     DX,OFFSET MAX
        INT     21H                     ;显示结果
        MOV     AH,4CH
        INT     21H                     ;返回 DOS
CODE     ENDS
        END     BEG

```

5.7 子程序及其调用

子程序是相对独立的程序,当程序中需要多次完成某一项操作的时候,为了简化整体程序和阅读方便,常常把完成某项操作的程序单独设计为一个子程序,需要时再调用它。

子程序通常使用 PROC/ENDP 做为定界语句,用 CALL 指令调用子程序,用 RET 指令返回。

从子程序所处的位置来分,有段内子程序和段间子程序。当子程序和调用它的主程序同在一个代码段时,子程序的属性应该定义为“NEAR”,否则应该定义为“FAR”。本小节仅介绍段内子程序调用,段间子程序调用在模块化程序设计一节中介绍。

子程序允许有多个入口,多个出口。子程序返回时通常是返回到调用程序断点,执行调用它的那条 CALL 指令的后续指令。如果需要,也可以不返回断点而返回到指定的位置。

子程序又分为无参数子程序和有参数子程序两种,使用有参数的子程序更加灵活。向子程序传送参数通常有三种方法:



- ① 利用寄存器传送参数,当要传送的参数较多时,这种方法不一定简单。
- ② 利用堆栈传送参数。
- ③ 利用内存单元传送参数。

下面例举一个加法程序,该程序完全可以不使用子程序调用,但是为了讲解子程序的返回方式以及子程序的参数传送技术,我们仍借用这个例题,用三种解法实现,请注意每一种解法的特点。三种解法均为段内调用,而且三种解法中的 DISP 子程序是三种不同的设计方法,请读者注意。

【例 5.7.1】 两字节加法程序。

假设: $N1=1122H$ 、 $N2=3344H$ 、 $N3=5566H$,计算并显示这三个数的累加和。

【解法 1 程序清单】

```

;FILENAME: 571_1. ASM
.486

DATA    SEGMENT USE16
NUM      DW      1122H          ;N1
          DW      3344H          ;N2
          DW      5566H          ;N3
DATA     ENDS
STACK_   SEGMENT STACK 'STACK' USE16
          DB      100 DUP(?)
STACK_   ENDS
CODE     SEGMENT USE16
          ASSUME  CS: CODE,DS: DATA,SS: STACK_
BEG:     MOV      AX,DATA
          MOV      DS,AX
          MOV      SI,OFFSET NUM      ;参数指针→SI
          CALL     COMPUTE
XYZ:     CALL     DISP
EXIT:    MOV      AH,4CH
          INT      21H
;-----
COMPUTE  PROC
          MOV      BX,0
          ADD      BX,[SI+0]          ;BX+N1→BX
          ADD      BX,[SI+2]          ;BX+N2→BX
          ADD      BX,[SI+4]          ;BX+N3→BX
          RET                          ;返回断点 XYZ
COMPUTE  ENDP
;-----
DISP     PROC                                ;显示 BX 内容
          MOV      CX,16
LAST:    MOV      DL,'0'
          RCL      BX,1

```

```

                JNC      NEXT
                MOV      DL,'1'
NEXT:          MOV      AH,2
                INT      21H
                LOOP     LAST
                RET                      ;返回断点 EXIT
DISP          ENDP
CODE          ENDS
                END      BEG

```

解法 1 设计了两个子程序, COMPUTE 子程序负责计算三个数的累加和, 结果送 BX 寄存器保存。DISP 子程序将 BX 寄存器的内容显示在屏幕上, 显然主程序和 DISP 子程序是使用 BX 寄存器传送参数的。

N1、N2、N3 在数据段, 调用 COMPUTE 子程序之前, 并没有把这些数据传送给子程序, 而是把存放这些数据的首址偏移量(即参数指针)赋给了 SI 寄存器, 进入 COMPUTE 子程序之后, 子程序用 SI 变址取数, 完成累加操作, 然后正常返回到调用程序的断点, 执行 DISP 子程序显示计算结果。主程序和 COMPUTE 子程序之间是采用内存单元传递参数的。

【解法 2 程序清单】

```

                ;FILENAME: 571_2. ASM
                .486
DATA          SEGMENT USE16
NUM           DW      1122H                ;N1
                DW      3344H                ;N2
                DW      5566H                ;N3
DATA          ENDS
STACK_        SEGMENT STACK 'STACK' USE16
                DB      100 DUP(?)
STACK_        ENDS
CODE          SEGMENT USE16
                ASSUME  CS: CODE, DS: DATA, SS: STACK_
BEG:          MOV      AX, DATA
                MOV      DS, AX
                MOV      SI, OFFSET NUM
                PUSH     WORD PTR [SI+0]      ;N1 压栈
                PUSH     WORD PTR [SI+2]      ;N2 压栈
                PUSH     WORD PTR [SI+4]      ;N3 压栈
                CALL     COMPUTE
XYZ:          CALL     DISP
EXIT:         MOV      AH, 4CH
                INT      21H
;-----
COMPUTE PROC

```



```

                MOV     BP,SP                ;栈指针→BP
                MOV     BX,0
                ADD     BX,[BP+2]            ;BX+N3→BX
                ADD     BX,[BP+4]            ;BX+N2→BX
                ADD     BX,[BP+6]            ;BX+N1→BX
                RET     6                    ;返回断点 XYZ,清栈
COMPUTE ENDP
;-----
DISP PROC                ;显示 BX 内容
                MOV     CX,16
LAST:  MOV     DL,'0'
                RCL     BX,1
                ADC     DL,0
                MOV     AH,2
                INT     21H
                LOOP    LAST
                RET                                ;返回断点 EXIT
DISP ENDP
CODE ENDS
END      BEG

```

解法 2 使用堆栈传送参数。编程者在程序的各个阶段应当熟知堆栈区域的参数变化。预先画出栈区参数的示意图,看图思索。

可执行程序装入之后,DOS 自动令 $SP=64H$,堆栈是“空”的。调用 COMPUTE 子程序之前使用了 3 条 PUSH 指令,依次把 N1、N2、N3 压入堆栈,然后执行 CALL COMPUTE 指令。进入 COMPUTE 子程序之后,栈顶存放的是断口地址 XYZ,如何访问堆栈中的 N1、N2、N3 呢? 子程序首先执行传送指令,把堆栈指针的当前值 SP 送 BP 寄存器,然后用 BP 寄存器访问堆栈中的 N1、N2、N3 完成累加操作。COMPUTE 子程序返回时,使用一条带参数的返回指令“RET 6”这是因为:子程序在完成累加操作之后,N1、N2、N3 仍然保留在堆栈中,如果不把它们“清除”掉就返回断点,堆栈中就多了 6 个元素,这是不规范的。一次调用,使堆栈中保留 6 个元素,10 次调用,堆栈中就累积了 60 个元素……,最终发生“堆栈溢出”产生不可预测的后果,因此程序员在使用堆栈时,必须保证进出平衡。怎样把 N1、N2、N3 清除掉呢? 使用 RET 6 指令,把栈顶元素弹到 IP,SP 下调 2 个单元之后,再下调 6 个单元,使 SP 仍旧等于 64H,从而恢复了堆栈的初始状态。

【解法 3 程序清单】

```

                ;FILENAME: 571_3. ASM
                .486
STACK_ SEGMENT STACK 'STACK' USE16
                DB      100 DUP(?)
STACK_ ENDS
CODE   SEGMENT USE16

```

```

                ASSUME  CS: CODE,SS: STACK_
BEG:            CALL    COMPUTE
NUM             DW      1122H                ;N1
                DW      3344H                ;N2
                DW      5566H                ;N3
XYZ:            CALL    DISP
EXIT:           MOV     AH,4CH
                INT      21H

;-----
COMPUTE PROC
                MOV     BP,SP                ;栈指针→BP
                MOV     SI,[BP+0]            ;断口地址 NUM→SI
                MOV     BX,0
                ADD     BX,CS:[SI+0]          ;BX+N1→BX
                ADD     BX,CS:[SI+2]          ;BX+N2→BX
                ADD     BX,CS:[SI+4]          ;BX+N3→BX
                POP     AX                    ;弹出原来的断口地址
                MOV     AX,OFFSET XYZ
                PUSH    AX                    ;新的返回地址压入栈顶
                RET                                ;返回断点 XYZ
COMPUTE ENDP

;-----
DISP            PROC                                ;显示 BX 内容
                MOV     CX,16
LAST:           MOV     AL,'0'
                RCL     BX,1
                ADC     AL,0
                MOV     AH,0EH
                INT      10H
                LOOP    LAST
                RET                                ;返回断点 EXIT
DISP            ENDP
CODE            ENDS
                END      BEG

```

解法 3 利用内存单元传送参数。在代码段 CALL COMPUTE 指令之后,紧接着用 DW 伪指令存放 N1、N2、N3 这 3 个参数。进入 COMPUTE 子程序之后,首先把堆栈指针 SP 的值转送给 BP,再用 BP 寻址取出栈顶单元的断口地址(即 NUM 的偏移地址)送 SI 寄存器,再用 SI 变址寻址,访问代码段中的 N1、N2、N3 完成累加操作。很明显,子程序执行结束后,不应当返回原来的断点 NUM,必须返回到 XYZ 才是合理的。如何返回新断点呢?本例采取的措施是“弃旧换新”,从栈顶弹出原来的断口地址,再压入新的返回地址,然后执行 RET 即可返回到指定的位置了。

5.8 宏指令与条件汇编

子程序调用可以简化程序设计,增强可读性。但是,如果子程序变量较多,调用子程序就比较麻烦。宏指令为我们简化程序设计开辟了另一个途径。

宏指令是程序员自己设计的指令,是若干指令的集合,用于完成某一项操作。即使是较低版本的宏汇编语言,也允许用户程序自定义宏指令,宏指令的定义语句可以不放在任何逻辑段之中,通常都放在程序的首部。

5.8.1 宏指令与宏调用

宏指令分为无参数宏指令与有参数宏指令两种:

1. 无参数宏指令的定义与调用

无参数宏指令的定义语句格式如下:

宏指令名称	MACRO	
	宏体	
	ENDM	
例如: CRLF	MACRO	
	MOV	AH,0EH
	MOV	AL,0DH
	INT	10H
	MOV	AL,0AH
	INT	10H
	ENDM	

MACRO/ENDM 是宏体的定界语句,上述宏体的功能是令光标返回到下一行的开始格,CRLF 是宏指令的名称。宏体经过定义之后,宏指令的名称 CRLF 就是一条宏指令,你可以像使用 CPU 指令那样去使用它,在代码段中放置一条 CRLF 就是宏调用。汇编时汇编程序用宏体替换宏指令,因此执行 CRLF 后,光标返回下一行始格。例如:

MOV	AH,2	
MOV	DL,'A'	
INT	21H	;在光标当前位置显示 A
CRLF		;光标返回下一行始格

2. LOCAL 伪指令

如果宏体中有分支、循环,必然有标号,两次以上调用这样的宏指令必然出现标号重复定义的错误。为此汇编语言提供了一条 LOCAL 伪指令可妥善解决这一问题。LOCAL 伪指令的格式如下:

使用说明:

- ① LOCAL 伪指令要放在宏定义之中,是 MACRO 定界语句以下的第一条语句。
- ② 标号名表是用逗号间隔的一串标号名,它们是宏体中出现的所有标号的集合。
- ③ 宏体中出现的标号称为局部标号,在使用了 LOCAL 伪指令之后,局部标号可以和源程序中的其他标号或变量重名。反之如果宏指令只被调用一次,而且宏体中的局部标号与源程序中的变量名或标号名都没有重复,那么宏体中可以不使用 LOCAL 伪指令。

有参数宏指令的定义语句格式如下:

上述格式中的哑元表,是一串用逗号间隔的形式参数表。哑元、形式参数是没有值的符号,用它(们)代表宏体中出现的操作码助记符、操作数(立即数、寄存器操作数、内存操作数),调用有参数宏指令的时候,宏指令行要有实元表,实元和哑元必须一一对应。实元可以是立即数、寄存器操作数以及没有 PTR 运算符的内存操作数,汇编时汇编程序按照一一对应的关系,把实参数赋给形式参数,再用宏体替换宏指令。调用有参数的宏指令大大简化了程序设计。它比调用有参数的子程序更方便,此外,宏指令还可以嵌套。例 5.8.1、例 5.8.2 给出了宏指令调用的示范,例 5.8.1 中 DISP 宏指令分组(4 位一组)显示 VAR 变量中的 NN 位二进制数,例 5.8.2 中 DISP 宏指令可在屏幕的指定位置显示一串彩色字符,这两个例题很有说服力,它充分说明使用有参数的宏指令是多么方便。

【例 5.8.1】 宏指令应用。

```

;FILENAME: 581.ASM
.486
CRLF MACRO                                ;光标返回下一行始格
MOV     AH,0EH
MOV     AL,0DH
INT     10H
MOV     AL,0AH
INT     10H
ENDM
DISP MACRO VAR,NN                        ;分组显示 VAR 中的 NN 位数
LOCAL  LAST1, LAST2
MOV     CH, NN/4
LAST1: MOV     CL, 4
LAST2: MOV     AL, '0'
        ROL     VAR, 1

```



```

        ADC      AL,0
        MOV      AH,0EH
        INT      10H
        DEC      CL
        JNZ      LAST2
        MOV      AL,' '
        INT      10H
        DEC      CH
        JNZ      LAST1
        CRLF
        ENDM
CODE    SEGMENT USE16
        ASSUME   CS: CODE
NUM     DB      88H
BEG:    MOV      EBX,12345678H
        DISP     BH,8           ;显示 BH 中的 8 位数
        DISP     BX,16          ;显示 BX 中的 16 位数
        DISP     EBX,32         ;显示 EBX 中的 32 位数
        DISP     NUM,8          ;显示 NUM 单元中的 8 位数
        MOV      AH,4CH
        INT      21H
CODE    ENDS
        END      BEG

```

【例 5.8.2】 定位显示彩色字符串。

```

;FILENAME: 582. ASM
. 486
DISP    MACRO    Y,X,VAR,LENGTH,COLOR
        MOV      AH,13H
        MOV      AL,1
        MOV      BH,0           ;选择 0 页显示屏
        MOV      BL,COLOR       ;属性字(颜色值)→BL
        MOV      CX,LENGTH      ;串长度→CX
        MOV      DH,Y           ;行号→DH
        MOV      DL,X           ;列号→DL
        MOV      BP,OFFSET VAR  ;串首字符偏移地址→BP
        INT      10H
        ENDM
EDATA   SEGMENT USE16
SS1     DB      'HELLO'
SS2     DB      'WELCOME !'
SS3     DB      'BYE_BYE'
EDATA   ENDS
CODE    SEGMENT USE16

```

```

ASSUME CS: CODE,ES: EDATA
MOV AX,EDATA
MOV ES,AX ;对 ES 初始化
MOV AX,3 ;设置 80×25
INT 10H ;彩色文本方式
DISP 0,5,SS1,5,2 ;0 行 5 列显示绿色 HELLO
DISP 12,36,SS2,9,4 ;12 行 36 列显示红色 WELCOME
DISP 24,66,SS3,7,0EH ;24 行 66 列显示黄色 BYE_BYE
SCAN: MOV AH,1
INT 16H
JZ SCAN ;等待用户键入,无键入则转移
MOV AX,2
INT 10H ;恢复 80×25 黑白文本方式
MOV AH,4CH
INT 21H
CODE ENDS
END BEG

```

4. 宏指令调用与子程序调用的区别

宏指令与子程序都可以简化程序设计,增强程序的可读性;子程序调用是由 CPU 完成的,宏指令调用是在汇编过程中由汇编程序完成的;子程序调用可以减小目标程序的体积,宏指令则不能,相反,频频调用宏指令只能使目标程序的字节数加大,虽然如此,因为计算机内存很大,还因为调用有参数的宏指令比调用有参数的子程序方便得多,因此在设计大型程序时,宏指令的使用非常广泛。

5.8.2 条件汇编

条件汇编语句是伪指令,它的功能是通知汇编程序,条件满足时汇编某些指令,否则就不汇编。条件汇编语句通常和逻辑运算符 AND、OR... 以及关系运算符 EQ、NE... 连用,构成条件。

1. 条件汇编语句的常用格式

格式 1: IF 条件	格式 2: IF 条件
指令集合 1	指令集合
ELSE	ENDIF
指令集合 2	
ENDIF	

IF/ENDIF 是一对定界语句,其中“条件”通常是逻辑表达式或者关系表达式。
 格式 1 功能: 如果条件成立仅汇编指令集合 1,不成立仅汇编指令集合 2。
 格式 2 功能: 如果条件成立,汇编指令集合,否则就不汇编指令集合。

2. 应用示范

假设源程序中定义了如下的宏指令：

```
SHIFT    MACRO    VAR,NN,LR
          IF      LR EQ 'L'
            ROL    VAR,NN
          ELSE
            ROR    VAR,NN
          ENDIF
        ENDM
```

代码段中执行以下程序段，结果如何？请看指令注解。

```
MOV      EBX,12345678H
PUSH     EBX
SHIFT    EBX,8,'L'           ;EBX=34567812H
POP      EBX
SHIFT    EBX,8,'R'           ;EBX=78123456H
```

5.9 代码转换

代码转换是程序设计的常见课题，由于键盘输入、屏幕显示、打印机输出都使用字符的 ASCII 码，而指令的运算对象及其结果都是一串 0、1 代码。输出显示的格式可能是二进制数、十进制数或十六进制数。因此在许多场合都要进行代码转换。

【例 5.9.1】 键盘输入的一位十进制数 ASCII 码→二进制数显示。

程序执行后，要求操作员键入 0~9 中的一个数，然后程序进行转换，显示出等值的二进制数。显示格式示范如下：

5=00000101B

当程序运行后，要求操作员临时给出执行参数的时候，有两种设计方法：一种是通过人机会话得到，另一种是通过命令行参数获取（本程序采用前一种方法）。程序在接收这些参数的时候，都应当有保护措施，以防止非法键入而引发的执行错误，这是设计高质量程序应具备的编程思想。

本程序采用两种格式设计源程序。一种是 EXE 文件编程格式，另一种是 COM 文件编程格式。图 5-7 是程序框图。

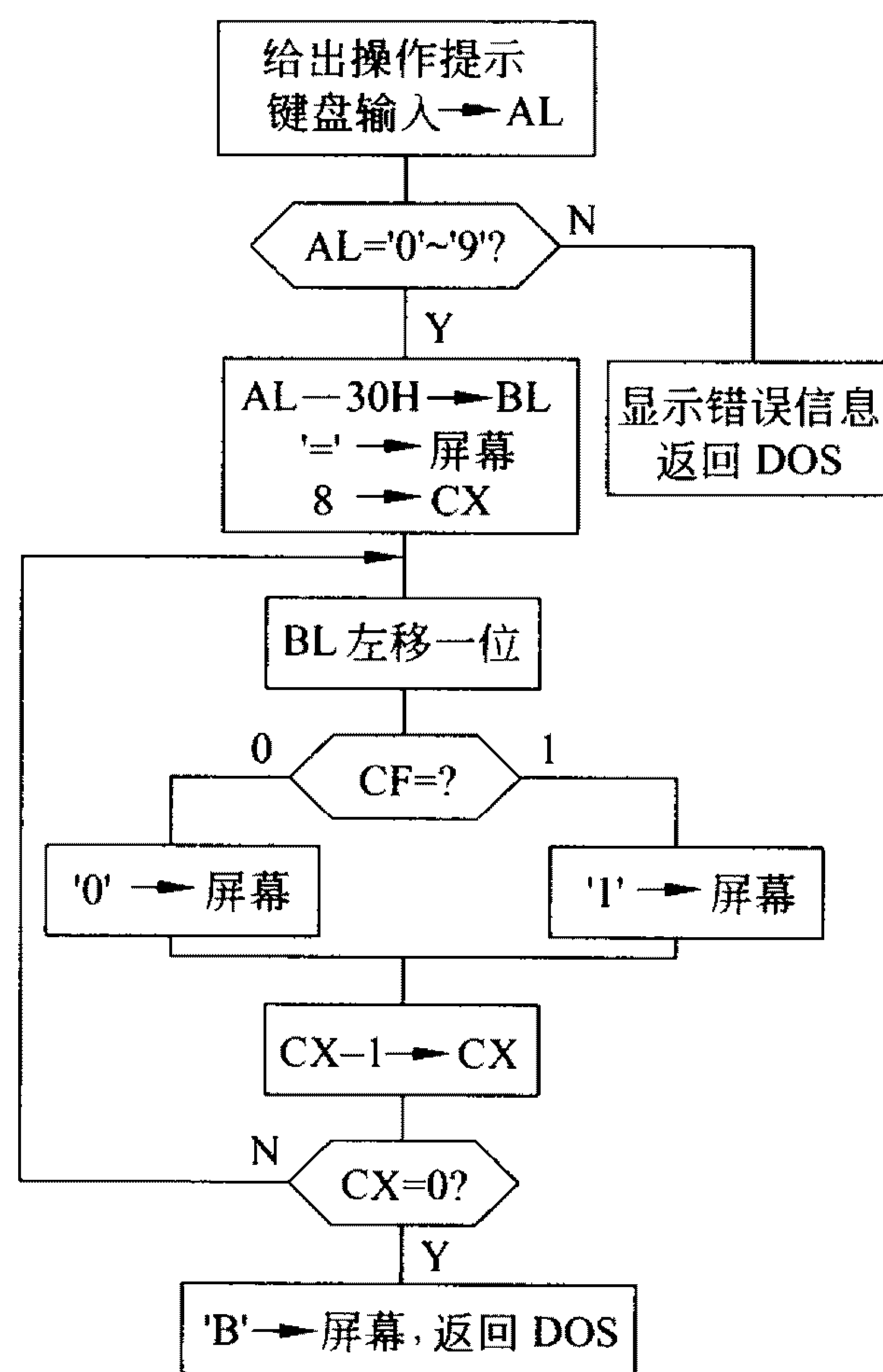


图 5-7 一位十进制数 ASCII 码→二进制数显示

【EXE 格式程序清单】

```

;FILENAME: 591_1. ASM
.486
DATA SEGMENT USE16
MSG1 DB 'Please Input! ',0DH,0AH,' $ '
MSG2 DB '---Error ! $ '
DATA ENDS
CODE SEGMENT USE16
ASSUME CS: CODE,DS: DATA
BEG: MOV AX,DATA
MOV DS,AX
MOV AH,9
MOV DX,OFFSET MSG1
INT 21H ;显示操作提示
MOV AH,1
INT 21H ;等待键入
CMP AL,3AH
JNC ERROR ;> '9' 转移
CMP AL,30H
JC ERROR ;< '0' 转移
SUB AL,30H
MOV BL,AL ;BL=0~9 的二进制数
MOV AH,2
MOV DL,'='
INT 21H
CALL DISP
MOV AH,2
MOV DL,'B'
INT 21H
JMP EXIT
ERROR: MOV AH,9
MOV DX,OFFSET MSG2
INT 21H ;显示错误信息
EXIT: MOV AH,4CH
INT 21H
;-----
DISP PROC ;显示 BL 中的二进制数
MOV CX,8
LAST: MOV DL,'0'
RCL BL,1
JNC NEXT
MOV DL,'1'
NEXT: MOV AH,2
INT 21H

```



```

        LOOP    LAST
        RET
DISP    ENDP
CODE    ENDS
        END     BEG

```

【COM 格式程序清单】

```

;FILENAME: 591_2. ASM
.486
CODE    SEGMENT USE16
        ASSUME  CS: CODE
        ORG     100H
START:  JMP     BEG
MSG1    DB      'Please Input! ',0DH,0AH,' $ '
MSG2    DB      '---Error ! $ '
BEG:    MOV     AH,9
        MOV     DX,OFFSET MSG1
        INT     21H                ;显示操作提示
        MOV     AH,1
        INT     21H                ;等待键入
        CMP     AL,3AH
        JNC     ERROR              ;> '9' 转移
        CMP     AL,30H
        JC      ERROR              ;< '0' 转移
        SUB     AL,30H
        MOV     BL,AL              ;BL=0~9 的二进制数
        MOV     AH,2
        MOV     DL,'='
        INT     21H
        CALL    DISP
        MOV     AH,2
        MOV     DL,'B'
        INT     21H
        JMP     EXIT
ERROR:  MOV     AH,9
        MOV     DX,OFFSET MSG2
        INT     21H                ;显示错误信息
EXIT:   MOV     AH,4CH
        INT     21H
;-----
DISP    PROC                ;显示 BL 中的二进制数
        MOV     CX,8
LAST:   MOV     DL,'0'
        RCL     BL,1

```

```

        JNC     NEXT
        MOV     DL,'1'
NEXT:   MOV     AH,2
        INT     21H
        LOOP    LAST
        RET
DISP    ENDP
CODE    ENDS
        END     START

```

【程序分析】

本例给出了两种编程格式,其中 COM 文件的编程格式只有一个逻辑段即代码段,数据也放在代码段之中。在代码段偏移地址为 100H 处放置了一条“JMP BEG”,以便跳过数据区。程序最后用“END START”伪指令结束,它通知汇编程序,目标程序的启动地址是 START。

就本例而言,COM 文件的体积为 110 个字节,而 EXE 文件的体积为 625 个字节,显然 COM 文件大大节省了磁盘空间。

【例 5.9.2】 二进制数 → 十六进制数显示。

设从 BNUM 单元开始有 4 个 16 位的二进制数,要求把它们转换成十六进制数并送屏幕显示。

【算法分析】

在文本方式下送往屏幕显示的任何字符都必须是该字符的 ASCII 码。

4 位二进制数其等值的十六进制数,及其相应的 ASCII 码有如下关系:

4 位二进制数: 0000,0001,...,1001,1010,...,1111

等值的十六进制数: 0 1 ... 9 A ... F

十六进制数 ASCII 码: 30H 31H ... 39H 41H ... 46H

从以上分析可知:当 4 位二进制数等于 0000~1001 时,该数加上 30H 就等于相应十六进制数的 ASCII 码;当 4 位二进制数等于 1010~1111 时,该数加上 37H 就等于相应十六进制数的 ASCII 码。因此在进行二进制→十六进制转换时,必须首先截取四位二进制数,然后判断其数值范围,再进行转换。

【程序清单】

```

;FILENAME: 592. ASM
.486
DATA    SEGMENT USE16
BNUM    DW      0001001000110100B      ;1234H
        DW      0101011001111000B      ;5678H
        DW      0001101000101011B      ;1A2BH
        DW      0011110001001101B      ;3C4DH
BUF     DB      4 DUP(?),'H $'         ;输出缓冲区
COUNT  DB      4
DATA    ENDS

```

```

CODE      SEGMENT USE16
          ASSUME  CS: CODE,DS: DATA
BEG:      MOV     AX,DATA
          MOV     DS,AX
          MOV     CX,4
          MOV     BX,OFFSET BNUM
AGAIN:    MOV     DX,[BX]           ;取一个数→DX
          SAL     EDX,16
          CALL    N2_16ASC         ;转换
          MOV     AH,9
          MOV     DX,OFFSET BUF
          INT     21H              ;显示一个 16 进制数
          ADD     BX,2             ;地址加 2
          LOOP    AGAIN
          MOV     AH,4CH
          INT     21H
;-----
N2_16ASC  PROC                ;二进制数→十六进制数 ACSII 码
          MOV     SI,OFFSET BUF   ;输出缓冲区地址→SI
          MOV     COUNT,4
LAST:     ROL     EDX,4
          AND     DL,0FH
          CMP     DL,10
          JC      NEXT
          ADD     DL,7
NEXT:     ADD     DL,30H
          MOV     [SI],DL
          INC     SI               ;地址加一
          DEC     COUNT           ;计数
          JNZ     LAST
          RET
N2_16ASC  ENDP
CODE      ENDS
          END     BEG

```

【例 5.9.3】 二进制数→十进制数显示。

二进制数转换成十进制数有多种编程方法,即:比较法、恢复余数法和除法求余等。

比较法最容易理解,我们以 8 位二进制数为例,描述它的算法。8 位二进制数其等值的十进制数最大为 255,比较法的编程技巧是首先求出该数包含几个 100,然后再求出余数包含几个 10、几个 1。比较法和恢复余数法程序框图如图 5-8 和图 5-9 所示,数据段 BEN 单元中为待转换的 8 位二进制数。

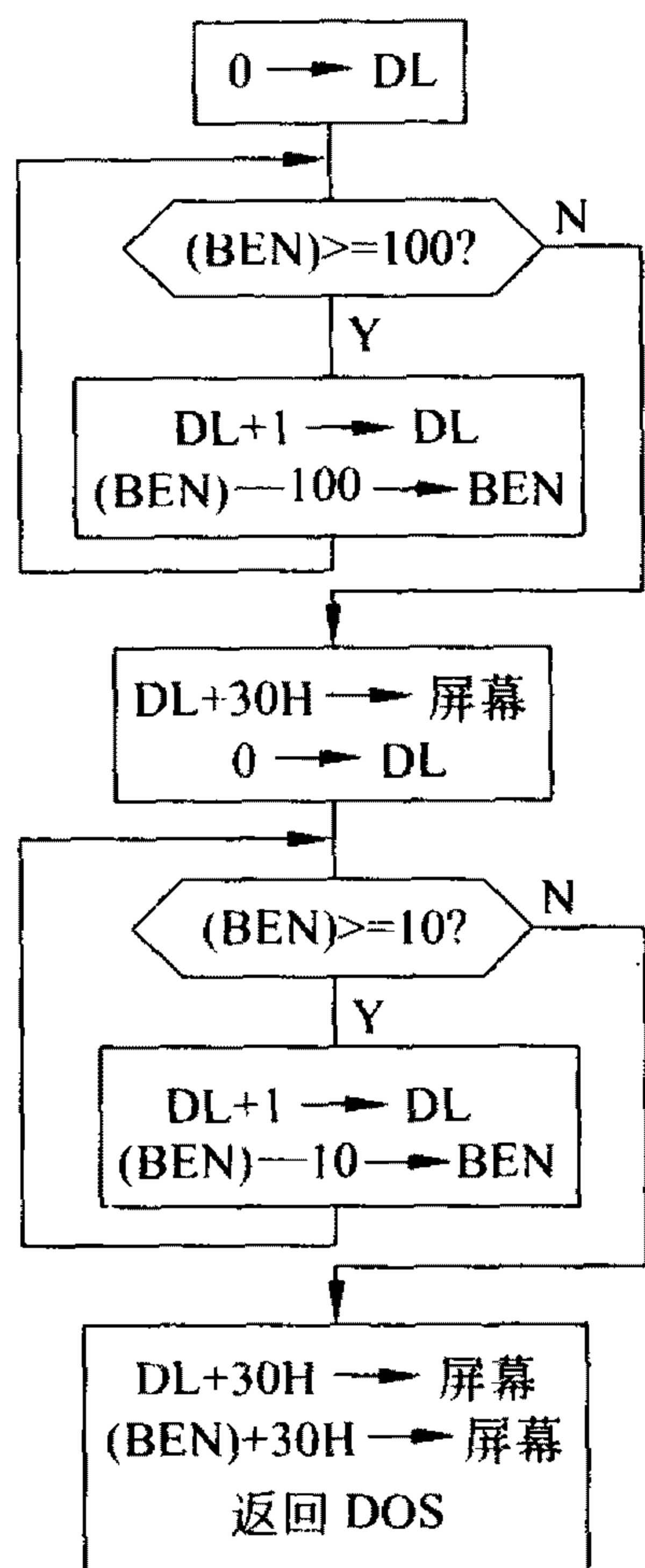


图 5-8 比较法程序框图

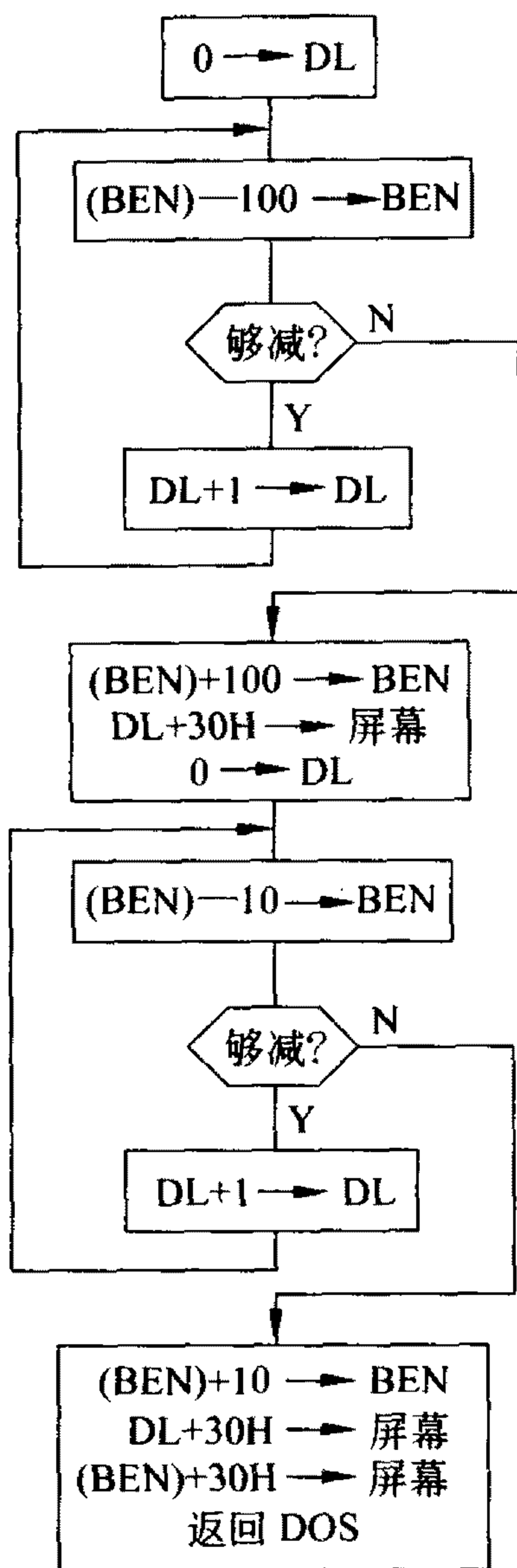


图 5-9 恢复余数法程序框图

【8 位二进制数转换比较法程序清单】

```

;FILENAME: 593_1. ASM
.486
CMPDISP MACRO NN
LOCAL LAST,NEXT
MOV DL,0
LAST:   CMP BEN,NN
JC NEXT
INC DL
SUB BEN,NN
JMP LAST
NEXT:   ADD DL,30H
MOV AH,2
INT 21H
ENDM
CODE    SEGMENT USE16
ASSUME CS: CODE
BEN     DB 10101110B
    
```

```

比较法

;DL 清零
;比较
;(BEN) < NN 转
;DL 加 1
;(BEN) - NN -> BEN 单元

;显示

;=174
    
```



```

BEG:      CMPDISP    100
          CMPDISP    10
          CMPDISP    1
          MOV        AH,4CH
          INT        21H
CODE      ENDS
          END        BEG

```

特例：一个小于等于 99 的二进制数，转换成十进制数显示更简单，请看下列程序：

```

          MOV        AX,01100010B      ;AH=0,AL=98
          AAM                          ;AH=09H,AL=08H
          ADD        AX,3030H          ;AH=39H,AL=38H
          PUSH       AX
          ROL        AX,8
          MOV        AH,0EH
          INT        10H                ;显示 9
          POP        AX
          MOV        AH,0EH
          INT        10H                ;显示 8

```

AAM 指令把 AX 中小于等于 99 (AH 要求为 0) 的二进制数转换成两个非组合的 BCD 码数→AH、AL，再对它们分别加 30H 送屏幕显示即可。

除法求余的编程思路是这样的：首先把待转换的二进制数除以 10，余数（必定小于 10）压入堆栈，再把商值除以 10，余数压栈……，直到商为 0 时为止！接下来依次从堆栈中弹出各次的余数，每弹出一个数就转换成 ASCII 码送屏幕显示。不论是 8 位、16 位还是 32 位二进制数程序不变。593_2. ASM 利用除法求余的算法，将 32 位二进制数转换成十进制数显示。

【32 位二进制数转换成十进制数显示除法求余程序清单】

```

          ;FILENAME: 593_2. ASM
          .486
CODE      SEGMENT USE16
          ASSUME    CS: CODE
NUM32     DD        3456789000      ;汇编后自动转换成二进制数
BEG:      MOV        EAX,NUM32
          MOV        EBX,10         ;除数 10→EBX
          MOV        CX,0           ;计数初值 0→CX
LAST:     MOV        EDX,0          ;0→EDX
          DIV        EBX            ;(EDX、EAX)÷EBX,商→EAX,余数→EDX
          PUSH       DX             ;余数压栈
          INC        CX             ;统计除法的次数
          CMP        EAX,0
          JNZ        LAST           ;商不为 0 转
AGA:      POP        DX             ;余数→DX

```

```

        ADD     DL,30H
        MOV     AH,2
        INT     21H           ;显示一位十进制数
        LOOP    AGA
        MOV     AH,4CH
        INT     21H
CODE    ENDS
        END     BEG

```

【例 5.9.4】 BCD 码→二进制数。

一位 BCD 码数用 4 位二进制数表示,其值在 0000~1001 之间,超出此范围是非法的 BCD 码数。在计算机中常用一位 BCD 码数代表一位 0~9 的十进制数。

本例要求把 BCD 字单元中的 4 位 BCD 码,转换成等值的二进制数显示。例如:

设 BCD 字单元中的数是: 0100 0101 0110 0111

等值的十进制数应当是: 4567

等值的二进制数应当是: 0001 0001 1101 0111

【算法分析】

假设 4 位 BCD 码数的数列为 N3 N2 N1 N0,其中 N3 为千位,N2 为百位,N1 为十位,N0 为个位。则:

等值的二进制数 = $N3 \times 1000 + N2 \times 100 + N1 \times 10 + N0 \times 1$

或 = $((N3 \times 10 + N2) \times 10 + N1) \times 10 + N0$

或 = $((((0 \times 10 + N3) \times 10 + N2) \times 10 + N1) \times 10 + N0$

以上算法的编程技巧是: 首先分离出 BCD 码的千位 N3、百位 N2、十位 N1、个位 N0,再设计一个完成 $AX \times 10 + BX \rightarrow AX$ 的子程序供调用。第 2 个算式的调用过程如下:

第 1 次调用时: 令 $AX = N3, BX = N2$,完成 $N3 \times 10 + N2 \rightarrow AX$ 。

第 2 次调用时: 令 $BX = N1$,完成 $(N3 \times 10 + N2) \times 10 + N1 \rightarrow AX$ 。

第 3 次调用时: 令 $BX = N0$,完成 $((N3 \times 10 + N2) \times 10 + N1) \times 10 + N0 \rightarrow AX$ 。

也可以按照第 1 个算式求累加和。下列程序按第 3 个算式进行 4 次调用。

【程序清单】

```

        ;FILENAME: 594. ASM
        . 486
CODE    SEGMENT USE16
        ASSUME  CS: CODE
BCD_NUM DW      4567H
BEG:    MOV     CX,4
        MOV     AX,0
AGAIN:  ROL     BCD_NUM,4
        MOV     BX,BCD_NUM
        AND     BX,000FH           ;依次截取千、百、十、个位 BCD 码→BX
        CALL    MUL10

```

```

                LOOP    AGAIN
DISP:          MOV     BP,AX
                MOV     CX,16
LAST:          MOV     AL,'0'
                RCL     BP,1
                ADC     AL,0
                MOV     AH,0EH
                INT     10H           ;依次显示 16 位二进制数
                LOOP    LAST
                MOV     AH,4CH
                INT     21H
MUL10          PROC           ;完成 AX×10+BX→AX
                MOV     DX,10
                MUL     DX
                ADD     AX,BX
                RET
MUL10          ENDP
CODE          ENDS
                END      BEG

```

特例：一个小于等于 99 的 BCD 码数可以直接用 AAD 指令求出其等值的二进制数。
例如：

```

MOV    AX,0908H    ;AH=9,AL=8,AX=BCD 码数 98
AAD                    ;AH×10+AL→AL,所以 AL=01100010

```

5.10 数值计算和数据处理

汇编语言中,可进行数值计算的仅有加、减、乘、除、移位等最基本的指令。运用这些基本指令完成稍微复杂一些的数值计算都是比较困难的。首先要探讨计算方法,要把某一问题分解成能够用加、减、乘、除完成的基本操作,然后才能着手编程。

数据处理涉及的面比较宽,其中字符串处理和表格处理都属于数据处理范畴。

数据表由若干表项组成。根据表项的内容可分为无序表和有序表两种。在无序表中各个表项的数值排列是无规则的,而在有序表中,表项是按其数值大小,从小到大,或从大到小依次排列的。

表格处理涉及数据查找、插入、删除、排序、数据表转换等操作,其中查表是最基本的操作。查表的方法有顺序查表、计算查表、对半搜索等。

【例 5.10.1】 求正整数 N 的开平方近似值。

【算法分析】

设：N=25,则 $N-1-3-5-7-9=0$,共减了 5 个奇数,再减 11 就不够减了。

N=40,则 $N-1-3-5-7-9-11=4$,共减了 6 个奇数,再减 13 就不够减了。

因此 N 的开平方近似值等于 N 依次减去前 i 个奇数,一直到不够减的时候为止,所减去的奇数个数就是 N 的开平方近似值。

我们设计一个子程序,假设 EAX 寄存器的内容为数 N ,由调用程序赋值,子程序返回后 ECX 寄存器的内容即为 N 的开平方近似值。

```

SQR    PROC                ;EAX=正整数 N
        MOV     ECX,0
LAST:   MOV     EDX,ECX
        ADD     EDX,EDX
        INC     EDX        ;EDX=前 i 个奇数
        SUB     EAX,EDX
        JC      EXIT
        INC     ECX
        JMP     LAST
EXIT:   RET
SQR    ENDP

```

【例 5.10.2】 为标准 ASCII 码设置校验位。

【题意】 计算机系统与 I/O 设备交换信息通常使用字符的 ASCII 码。最高位(D_7 位)为 0 的 ASCII 码称为标准 ASCII 码,最高位(D_7 位)为 1 的 ASCII 码称为扩展 ASCII 码。在字符传送过程中,为了使收方能够判断出接收字符是否正确,最简易的方法是定义标准 ASCII 码的最高位为奇偶校验位,包括校验位在内的一个字节中“1”的个数为奇数,则称为奇校验 ASCII 码,反之一个字节中“1”的个数为偶数则称为偶校验 ASCII 码。校验位的值由发送方设置,接收方负责校验,如果双方约定是偶校验传输,而接收方收到的一个字符编码中,有奇数个“1”,那么这个字符肯定是错误的。带校验的传输是计算机通信中,识别传输错误的最简单的方法。

本例题仅要求把 STRING 单元中的一串标准 ASCII 码,转换成奇校验 ASCII 码存入 BUF 单元开始的缓冲区,下面用两种风格编程。

【解法 1 程序清单】

```

                ;FILENAME: 5102_1.ASM
                .486
DATA    SEGMENT USE16
STRING  DB      'THE QUICK BROWN FOX JUMPS OVER LAZY DOG'
COUNT  EQU     $-STRING        ;统计串长度
DATA    ENDS
EXTRA   SEGMENT USE16
BUF     DB      COUNT DUP(?)    ;预留等长的缓冲区
EXTRA   ENDS
CODE    SEGMENT USE16
        ASSUME  CS: CODE,DS: DATA,ES: EXTRA
BEG:    MOV     AX,DATA
        MOV     DS,AX            ;DS 初始化

```



```

MOV     AX,EXTRA
MOV     ES,AX           ;ES 初始化
MOV     CX,COUNT
MOV     SI,OFFSET STRING ;原串首址→SI
MOV     DI,OFFSET BUF    ;目标区首址→DI
CLD                     ;D 标志为 0,增址型
LOAD:   LODSB           ;DS: [SI] →AL
        AND     AL,AL
        JNP     STORE   ;有奇数个“1”转
        OR      AL,80H   ;否则置校验位为 1
STORE:  STOSB           ;AL→ES: [DI]
        LOOP    LOAD
        MOV     AH,4CH
        INT     21H
CODE    ENDS
        END     BEG

```

【解法 2 程序清单】

```

;FILENAME: 5102_2. ASM
.486
DATA    SEGMENT USE16
STRING  DB      'THE QUICK BROWN FOX JUMPS OVER LAZY DOG '
COUNT  EQU     $-STRING
BUF      DB      COUNT DUP(?)
DATA     ENDS
CODE     SEGMENT USE16
        ASSUME  CS: CODE,DS: DATA,ES: DATA
BEG:     MOV     AX,DATA
        MOV     DS,AX
        MOV     ES,AX           ;DS=ES=DATA
        MOV     CX,COUNT
        MOV     SI,OFFSET STRING
        MOV     DI,OFFSET BUF
        CLD
LOAD:    LODSB
        AND     AL,AL
        JNP     STORE
        OR      AL,80H
STORE:   STOSB
        LOOP    LOAD
        MOV     AH,4CH
        INT     21H
CODE     ENDS
        END     BEG

```

【程序分析】

程序使用串装入指令 LODSB 从 STRING 单元取一个数装入 AL 再判断其奇偶性,当取出的一个字节数据,有偶数个“1”的时候,置最高位为 1(否则不处理),然后用串存储指令 STOSB 存入 BUF 单元……。

LODSB 指令要求源串在数据段,而 STOSB 指令要求目标区在 ES 附加段,因此解法 1 共设置了 3 个逻辑段。ASSUME 语句中用“ES: EXTRA”说明以 EXTRA 为段名的是附加段。

解法 2 是另一种编程风格,它设置数据段和附加段“重叠”。怎样令数据段和附加段重叠呢?要采取两项措施:其一,在 ASSUME 语句中用“DS: DATA”和“ES: DATA”说明以 DATA 为段名的逻辑段既是数据段,又是 ES 附加段;其二,在程序一开始给 DS 和 ES 赋相同的段基址。

设置段重叠有什么好处呢?它可以使程序员随心所欲地使用串操作指令,而不必顾及对源串和目标区设置的段约定,减小了出错的概率。我们在以后的程序中,如果涉及串操作指令一律采用段重叠的编程风格。

【例 5.10.3】 数据查找。

设内存缓冲区从 BUF+1 单元开始,有若干个单字节有符号数,其个数存放在 BUF 单元中,要求找出最大数送 MAX 单元,最小数送 MIN 单元。

【编程思路】 数据查找的关键是进行数据比较,对于有符号数查找,其最大数和最小数都是针对真值而言的,因此应使用有符号数的比较转移指令。

【程序清单】

```
                ;FILENAME: 5103. ASM
                . 486
DATA            SEGMENT USE16
BUF             DB      8,34,56,-1,7FH,-88,200,22,80H
MAX             DB      ?
MIN             DB      ?
DATA            ENDS
CODE            SEGMENT USE16
                ASSUME  CS: CODE,DS: DATA
BEG:            MOV     AX,DATA
                MOV     DS,AX
                MOV     CH,0
                MOV     CL,BUF                ;CX=数据个数
                DEC     CX                    ;CX=比较次数
                MOV     AL,BUF+1
                MOV     MAX,AL                ;假设第一个数是最大数
                MOV     MIN,AL                ;假设第一个数是最小数
                MOV     BX,OFFSET BUF+2
LAST:           MOV     AL,[BX]
                CMP     AL,MAX                ;比较
```



```

                JG      GREAT
                CMP     AL,MIN
                JL      LESS
                JMP     NEXT
GREAT:          MOV     MAX,AL          ;大数→MAX
                JMP     NEXT
LESS:           MOV     MIN,AL          ;小数→MIN
NEXT:           INC     BX
                LOOP    LAST
                MOV     AH,4CH
                INT     21H
CODE            ENDS
                END     BEG

```

【例 5.10.4】 命令行参数的传递技术。

将命令行参数传递到用户程序 BUF 开始的数据区。

【基本概念与设计思路】

在程序执行过程中,需要临时输入数据时,有两种方法:一种是通过人机会话(例 5.2.1),另一种是设置命令行参数然后传递给执行程序,例如第 4 章介绍的 TASM.EXE,TLINK.EXE 程序。

在 DOS 提示符下键入一个可执行文件的文件名时,文件名之后可以携带参数,参数与文件名之间最少有一个空格(即间隔符),多键入几个空格也是允许的。但是 DOS 的专用符,即: I/O 改向符“<”、“>”和管道操作符“|”不可以做参数,命令行以回车键做为结束标志。

怎样把命令行参数传递给用户程序呢? 必须熟知程序段前缀的数据结构。我们在第 5 章“程序段前缀”一节中介绍过: DOS 在装载一个可执行文件时,自动在其上方低地址处建立一个有 256 个字节的程序段前缀(PSP),并且:

① DOS 把命令行之后从第一个间隔符开始(包括这个间隔符)到回车键(包括回车键)之间的所有字符编码传送到 PSP: 81H 开始的内存单元。

② DOS 统计第一个间隔符到回车键(不包括回车键)之间的字符个数,并把统计结果写入 PSP: 80H 单元(其中符号 PSP 代表程序段前缀的段基址)。

③ 对于 EXE 文件,DOS 自动令 DS=ES=程序段前缀的段基址,自动令 CS: IP 指向用户程序的启动指令,然后才转而执行 CS: IP 处的指令。

编程的关键问题是: 程序一开始不能破坏 DS、ES 中的 PSP 段基址,首先删除有效参数之前的间隔符,然后再把有效参数传递到用户程序指定的缓冲区,供后续程序(本例无)使用。我们在 5104. ASM 程序中增加了一段显示程序,目的是验证一下传递过来的参数是不是命令行的有效参数。

【程序清单】

```
;FILENAME: 5104. ASM
```

```
. 486
```

```

DATA      SEGMENT  USE16
BUF        DB      127 DUP(?)
           DB      '$'

DATA      ENDS
CODE      SEGMENT  USE16
           ASSUME   CS: CODE, DS: DATA, ES: DATA
BEG:       MOV      CL, DS: [80H]           ;DS=PSP 段基址
           MOV      CH, 0                  ;CX=命令行参数的长度
           MOV      SI, 81H
LAST:      CMP      BYTE PTR [SI], ' '     ;命令行当前字符是空格?
           JNE      NEXT                  ;不是转移
           INC      SI
           DEC      CX                     ;以上程序的功能是
           JMP      LAST                  ;删除有效参数前的空格
NEXT:      MOV      AX, DATA
           MOV      ES, AX
           MOV      DI, OFFSET BUF
           CLD                             ;将格式化后的命令行参数
           REP      MOVSB                 ;传送到用户程序附加段
           CALL     DISP                  ;显示格式化后命令行参数
           MOV      AH, 4CH
           INT      21H
DISP       PROC
           MOV      AX, DATA
           MOV      DS, AX
           MOV      AH, 9
           MOV      DX, OFFSET BUF
           INT      21H
           RET
DISP       ENDP
CODE      ENDS
           END      BEG

```

【例 5.10.5】 删除数据项。

设 NUM 单元为数据个数, NUM+1 单元开始是一张无序表, 请查找 NN 单元的数是否在数列之中, 若是则从数列中删去该数。

【设计思路】

对于无序表, 应从第一个数开始依次搜索, 当找到需要删除的数据之后, 只需将后续的数据块依次向低地址方向移动一个字节, 将需要删除的数据覆盖掉, 就完成了删除任务。使用字符串搜索指令和字符串传送指令比较简单, 问题是有些字符串操作指令需要设置附加段。当程序中既要求有数据段, 又要求有附加段的时候, 为了简化设计, 通常的做法是不单独设置附加段, 而是定义数据段和附加段重叠。怎样定义数据段和附加段重叠? 在 ASSUME 语句中说明 DS 和 ES 寻址同一个逻辑段, 在其后的赋值语句中, 给 DS

和 ES 赋予同一个逻辑段的段基址,即可达到设置段重叠的目的。本程序采用这种编程风格。为了验证,本程序另设计一个 DISP 子程序,用来显示删除操作之后的数据表内容。

【程序清单】

```

;FILENAME: 5105. ASM
. 486
DATA SEGMENT USE16
NUM DB 9,'A1B2C3E45'
NN DB 'E'
DATA ENDS
CODE SEGMENT USE16
ASSUME CS: CODE,DS: DATA,ES: DATA
BEG: MOV AX,DATA
MOV DS,AX
MOV ES,AX
MOV AL,NN ;关键字→AL
MOV CH,0
MOV CL,NUM ;数据串长度→CX
MOV DI,OFFSET NUM+1 ;串首址→ES: DI
CLD
REPNE SCASB ;搜索关键字
JNZ EXIT ;没有关键字转移
MOV SI,DI
DEC DI
REP MOVSB ;后续数据块上移一个单元
DEC NUM ;串长度减一
CALL DISP
EXIT: MOV AH,4CH
INT 21H
DISP PROC
MOV NN-1,'$'
MOV AH,9
MOV DX,OFFSET NUM+1
INT 21H
RET
DISP ENDP
CODE ENDS
END BEG

```

【例 5.10.6】 字符串搜索。

假设从 STRING 单元开始有一源串字符,程序执行后采用人机会话方式,从键盘输入一个任意长度的子串,请查找源串中是否蕴含着键入的子串,并给出结果信息。

【设计思路】

当子串长度为 0,或者超出源串长度的时候没有必要搜索,只有子串的长度小于或者等于源串长度时才进行搜索。搜索的次数应当是源串长度减去子串长度再加 1。

我们用 DOS 系统 0AH 号功能调用,接受键入的子串,并限制子串的有效长度(不包括回车)不超过源串长度,然后使用带有重复前缀 REPE 的字节串比较指令进行比较。

【程序清单】

```

;FILENAME: 5106. ASM
. 486
DISP      MACRO      VAR
MOV       AH,9
MOV       DX,OFFSET VAR
INT       21H
ENDM

DATA      SEGMENT USE16
STRING    DB          'BASIC FORTRAN_77 C++ FOXPRO JAVA'
LENS      EQU         $ - STRING          ;源串长度
BUF       DB          LENS+1              ;限制子串长度最长为 LENS
          DB          ?
          DB          LENS+1 DUP(?)      ;预留子串的存储空间
NNN       DW          0
COUNT    DB          ?
MSG1      DB          'Please Input... $ '
MSG2      DB          0AH,'--- Found ! $ '
MSG3      DB          0AH,'--- Not Found ! $ '
DATA      ENDS

CODE      SEGMENT USE16
ASSUME    CS: CODE,DS: DATA,ES: DATA
BEG:      MOV         AX,DATA
          MOV         DS,AX
          MOV         ES,AX
          DISP        MSG1
          MOV         AH,0AH
          MOV         DX,OFFSET BUF
          INT         21H                ;接收一个子串
          MOV         AL,BUF+1
          CMP         AL,0
          JZ          NOFOUND           ;子串长度为 0 转移
          MOV         BYTE PTR NNN,AL
          MOV         AH,LENS
          SUB         AH,AL
          INC         AH
          MOV         COUNT,AH          ;搜索次数→COUNT 单元
          CLD

```

```

MOV     BX,OFFSET STRING
AGAIN:  MOV     SI,BX
        MOV     DI,OFFSET BUF+2
        MOV     CX,NNN
        REPE    CMPSB           ;字符串搜索
        JZ      FOUND
        INC     BX
        DEC     COUNT
        JNZ     AGAIN
NOFOUND: DISP   MSG3
        JMP     EXIT
FOUND:  DISP   MSG2
EXIT:   MOV     AH,4CH
        INT     21H
CODE    ENDS
END     BEG
```

【例 5.10.7】 找小元排序。

假设内存中从 BUF 单元开始有若干单字节无符号数,要求把它们按其数值大小,从小到大重新排列。

排序是数据处理中较为常见的问题。所谓排序,就是对若干毫无规则的数据进行整理,按其数值大小,从小到大或者从大到小重新排列,使它们成为一组有序的数列,为以后的快速查询、数据增删做准备。

【算法分析】

① 排序的关键性措施是进行数的比较。参看表 5-2,假设从 BUF 单元开始有 4 个无符号数,我们设置一个交换标志(FLAG 单元),每一轮比较开始,首先令交换标志 FLAG=0,然后进行若干次相邻两单元的数的比较,第一轮共进行 3 次比较,即比较 BUF 和 BUF+1 单元的数,比较 BUF+1 和 BUF+2 单元的数,比较 BUF+2 和 BUF+3 单元的数。每次比较之后,把小数存入低地址单元,大数存入高地址单元。只要发生一次数据交换,就令交换标志 FLAG=1,如果没有进行数据交换,则 FLAG 应保持不变。第一轮比较之后,最大数 44 存入数组的末地址单元,它没有必要再参加下一轮比较;

表 5-2 找小元排序算法分析

	第一轮比较 令 FLAG 单元=0			第二轮比较 令 FLAG 单元=0		第三轮比较 令 FLAG 单元=0
比较次数	1	2	3	1	2	1
BUF:44	22			22		11
22	44	33		33	11	22
33		44	11		33	
11			44			
	令 FLAG 单元=1			令 FLAG 单元=1		令 FLAG 单元=1

② 每一轮比较结束,检查交换标志,若 $FLAG=1$ 进行下一轮比较, $FLAG=0$ 表示排序结束;

③ N 个数,第一轮比较 $N-1$ 次,第二轮比较 $N-2$ 次……。

从上述算法分析示意图中看出,有两种情况表示排序结束。

其一:每一轮比较结束,若交换标志 $FLAG=0$,表示排序结束。

其二: N 个数在进行了 $N-1$ 轮比较之后,虽然交换标志 $FLAG=1$,但数据已经按照数值递增的规律排列好了,没有必要再进行下一轮比较,即 N 个数最多进行 $N-1$ 轮比较即可完成排序。图 5-10 是本例程序框图。

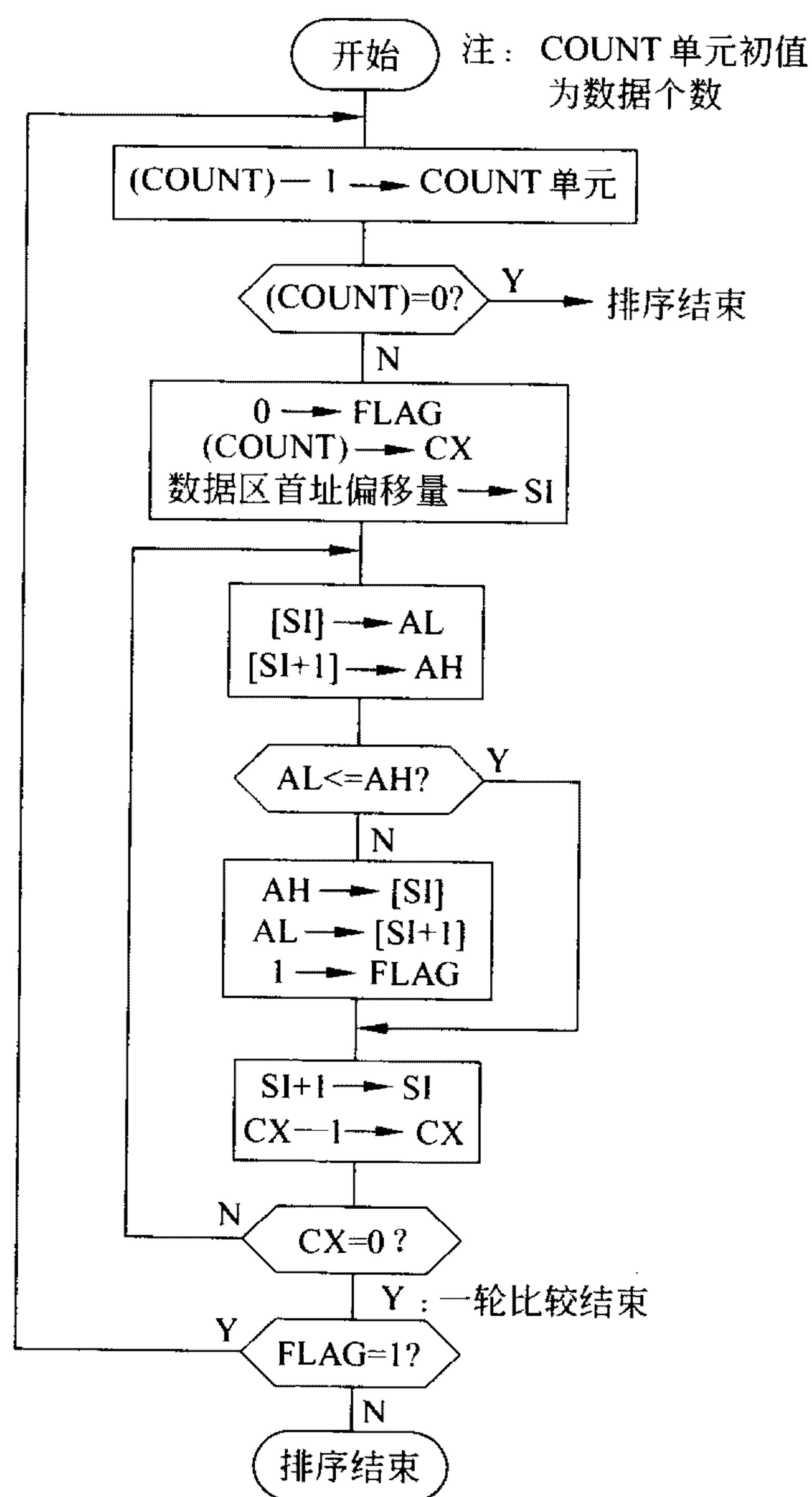


图 5-10 找小元排序程序框图

【程序清单】

```

;FILENAME: 5107.ASM
.486
DATA SEGMENT USE16
BUF DB 'ASDFGUYTNBV7654PLKM'
LENS EQU $-BUF ;LENS 等于数据个数

```



```

COUNT    DW      LENS
FLAG       DB      ?
DATA       ENDS
CODE       SEGMENT USE16
            ASSUME  CS: CODE,DS: DATA
BEG:       MOV     AX,DATA
            MOV     DS,AX
AGAIN:     DEC     COUNT
            JZ      DONE                ;排序结束转
            MOV     FLAG,0              ;置交换标志为 0
            MOV     CX,COUNT            ;每一轮的比较次数→CX
            MOV     SI,OFFSET BUF
LAST:      MOV     AL,[SI]
            MOV     AH,[SI+1]
            CMP     AH,AL
            JNC     NEXT
            MOV     [SI],AH             ;数据交换
            MOV     [SI+1],AL           ;数据交换
            MOV     FLAG,1              ;置交换标志为 1
NEXT:      INC     SI
            LOOP    LAST
            CMP     FLAG,1              ;若交换标志为 1
            JE      AGAIN               ;进行下一轮比较
DONE:      MOV     BUF+LENS,' $ '
            MOV     AH,9
            MOV     DX,OFFSET BUF
            INT     21H
            MOV     AH,4CH
            INT     21H
CODE       ENDS
            END     BEG

```

【例 5.10.8】 对半搜索。

假设从 BUF 单元开始的数据区是有序数列,数据已经按照数值递增的规律排列好了,请检查 KEY 单元中的数(该数假设由键盘输入)是否在数据区中。若是则显示“Found”,否则显示“No Found!”。

数据搜索就是查询数据区,检测是否有需要的数据(该数据通常称为“关键字”),对半搜索是速度较快的一种方法。

对半搜索的前提是数据有序,即:数据已经按照数值递增的规律,或者数值递减的规律排列好了,如果数据是杂乱无章的无序排列,只能用线性搜索逐个比较的方法查找。本例假设的数据区是递增数列。

【算法分析】

以图 5-11 为例,假设 100~106 单元中的数据是数值递增的数据。

- ① 首先取数组的中间数和关键字比较。怎样得到中间数呢？令：

搜索地址 = (搜索区上址 + 搜索区下址) ÷ 2 取整

 则搜索地址单元中的内容就等于当前搜索区的中间数。
- ② 如果关键字大于中间数，说明关键字可能在下区，即搜索地址和当前搜索区的末地址之间。然后用搜索地址取代上地址，再次求出新的搜索地址……。
- ③ 如果关键字小于中间数，表示关键字可能在上区，即当前搜索区的首地址和搜索地址之间。这样就需要用搜索地址取代下地址，再次求出新的搜索地址……。
- ④ 怎样才能搜索到数组中的任意一个数呢？在首次计算搜索地址的时候，应当令：

搜索区上址 = 数据区首址 搜索区下址 = 数据区末址 + 1
- ⑤ 何时结束搜索？通过以上分析可知，当中间数等于关键字的时候，表示关键字已经找到，即可结束搜索。当中间数不等于关键字，而且搜索地址等于当前搜索区的上地址的时候，表示数组中没有关键字，也应结束搜索。图 5-11 是本例的程序框图。

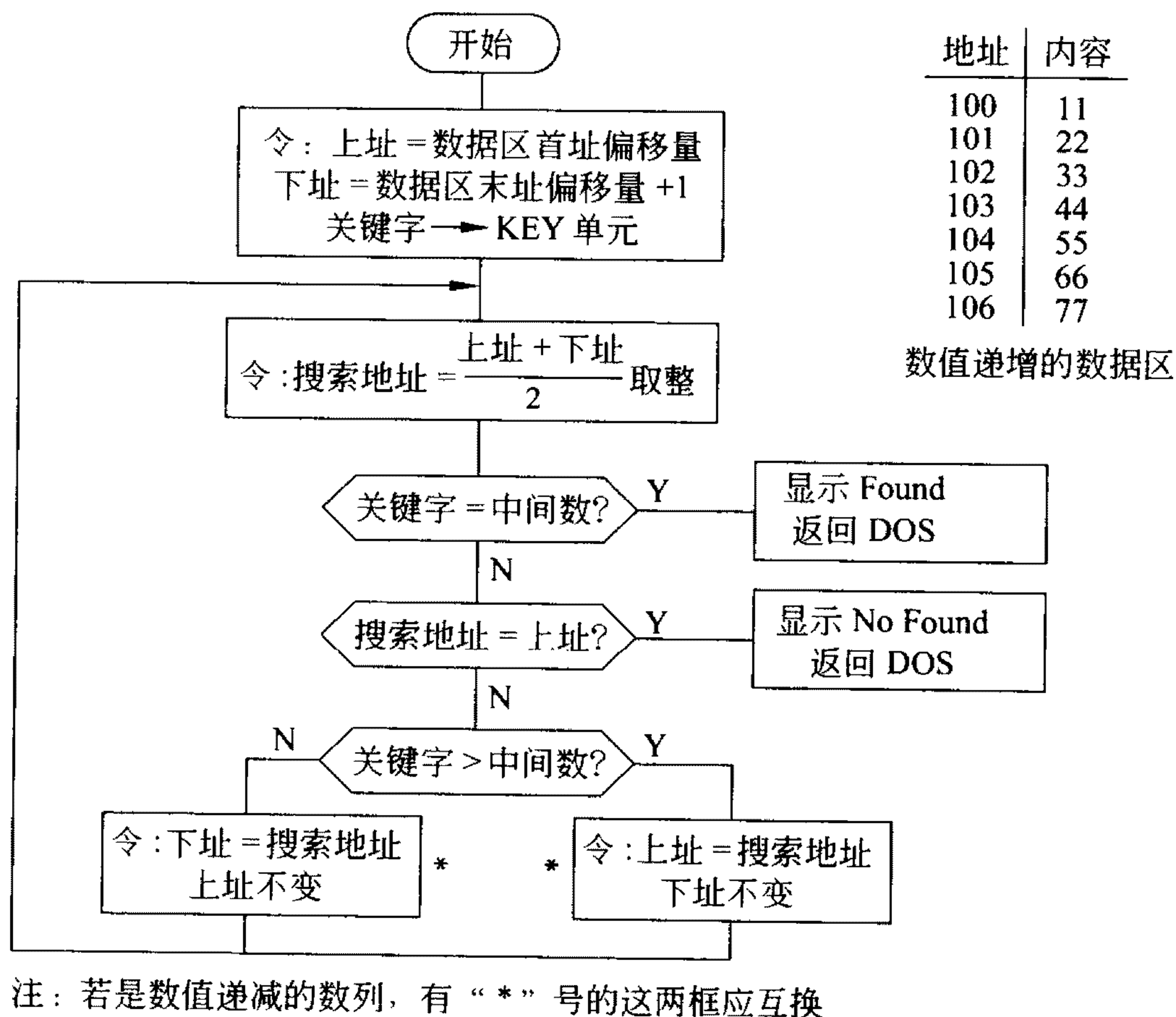


图 5-11 对半搜索程序框图

【程序清单】

```

;FILENAME: 5108. ASM
.486
DISP  MACRO  VAR
MOV    AH,9
MOV    DX,OFFSET VAR
INT    21H
ENDM
DATA  SEGMENT  USE16

```

```

BUF      DB      'ABCDEFGHIJKLMNOPQRSTUVWXYZ' ;递增数列
COUNT   EQU     $ - BUF
KEY       DB      ?
LAST      DW      BUF                          ;上址
NEXT      DW      BUF + COUNT                  ;下址
MSG       DB      'Input... $ '
OK        DB      '---Found ! $ '
NO        DB      '---Not Found ! $ '
DATA      ENDS
CODE      SEGMENT USE16
          ASSUME   CS: CODE, DS: DATA
BEG:      MOV      AX, DATA
          MOV      DS, AX
          DISP     MSG
          MOV      AH, 1
          INT      21H
          MOV      KEY, AL                      ;关键字→KEY 单元
AGAIN:    MOV      BX, LAST
          ADD      BX, NEXT
          RCR      BX, 1                        ;BX=搜索地址
          MOV      AL, KEY
          CMP      AL, [BX]                    ;关键字=中间数?
          JE       FOUND                      ;等于转移
          CMP      BX, LAST                    ;搜索地址=上址?
          JE       FAIL                       ;等于转移
          CMP      AL, [BX]                    ;关键字>中间数?
          JNC      GREAT                      ;大于转移
          MOV      NEXT, BX                    ;令下址=搜索地址
          JMP      AGAIN
GREAT:    MOV      LAST, BX                    ;令上址=搜索地址
          JMP      AGAIN
FOUND:    DISP     OK
          JMP      EXIT
FAIL:     DISP     NO
EXIT:     MOV      AH, 4CH
          INT      21H
CODE      ENDS
          END      BEG

```

【例 5.10.9】 延时程序。

在系统应用中,有时需要在执行某一操作之后,延时一段时间再进行下一个操作,为了实现延时可以采用软件、硬件或者软硬件结合的措施,本节仅介绍用软件实现延时的方法。

(1) 最简单的延时程序

```
        MOV     EBP,立即数 N
LAST:   DEC     EBP
        JNZ     LAST
```

该程序利用指令的多次循环,消耗 CPU 一段时间,从而达到延时的目的,改变立即数 N 的大小,就改变了循环次数。系统主频越高,延时时间就越短。把上述程序修改成双循环结构可增加延时的时间。同一个延时程序在主频不同的计算机中,延时时间也不同,因此只能在那些对延时时间要求极不精确的场合下使用。

(2) 利用 BIOS 功能设计延时程序

【INT 15H 86H 子功能】延时。

入口参数: CX=延时时间的高 16 位微秒值。

DX=延时时间的低 16 位微秒值。

出口参数: C 标志=1 预置失败, C 标志=0 预置成功。

该项功能调用不破坏系统时间,一次只供一个进程使用,举例如下:

```
MOV     AH,86H
MOV     CX,0
MOV     DX,50000      ;延时时间 50000 $\mu$ s $\rightarrow$ DX
INT     15H
```

80286 以上的高档微型计算机利用 INT 15H 软中断,增加了 86H 子功能,调用该功能,在 DOS 环境下以及 Windows 98(含 98)以下的 MS-DOS 方式均可实现精确延时的目的。实验证明:这种延时功能在 Windows 2000(含 2000)以上操作系统下的“命令提示符”方式,用户程序无法成功地调用 INT 15H 的 86H 子程序,不能起到延时的作用。

(3) 利用 DOS 系统功能设计延时程序

DOS 系统 2CH 号子程序:读取系统时间,2DH 号子程序:预置系统时间,利用这两个子程序可设计比较精确的延时程序。

【INT 21H 2CH 子功能】读取系统时间。

入口参数:无。

出口参数: CH=当前时间的小时数 (0~23 的二进制数)。

CL=当前时间的分钟数 (0~59 的二进制数)。

DH=当前时间的秒数 (0~59 的二进制数)。

DL=当前时间的百分秒 (0~99 的二进制数)。

【INT 21H 2DH 子功能】预置系统时间。

入口参数: CH=小时 (0~23 的二进制数)。

CL=分钟 (0~59 的二进制数)。

DH=秒 (0~59 的二进制数)。

DL=百分秒 (0~99 的二进制数)。

出口参数: AL=0 预置成功, AL=FFH 预置失败,失败原因是预置的时间值中至少有一项是无效数据,例如: 25 小时或 63 秒。

调用 2CH,2DH 号子功能设计的延时子程序如下:

【延时子程序】

```

DELAY      PROC                                ;延时子程序
            MOV     AH,2DH
            MOV     CX,0
            MOV     DX,0
            INT     21H
READ:      MOV     AH,2CH
            INT     21H
            MOV     AL,100
            MUL     DH
            MOV     DH,0
            ADD     AX,DX
            CMP     AX,SI                      ;SI 存放欲延时的时间(百分秒)
            JC      READ
            RET
DELAY      ENDP

```

该程序在 DOS 环境下以及 Windows 95, Windows 98 的 MS_DOS 方式, Windows 2000, Windows 2000XP 环境下的“命令提示符”方式都能起到延时作用。

5.11 字符串的动态显示技术

应用程序一般都有一个标题,或者是几行简易的使用说明,如果字符串能够动态地显示在屏幕上,一定会令使用者欣慰。字符串的动画显示,有左移位显示,右移位显示。而左右对进的移位显示动感最强。

【例 5.11.1】 用左右对进的移位显示方法,动态地显示一个应用程序的标题栏,标题栏静止后,显示格式如图 5-12 所示,边框正中的使用说明为: 串行口测试程序。



图 5-12 标题栏信息

【字符串动画显示的原理】

字符串动画显示的原理,就是移位显示和延时这两项操作的交替使用。如图 5-13 所示:

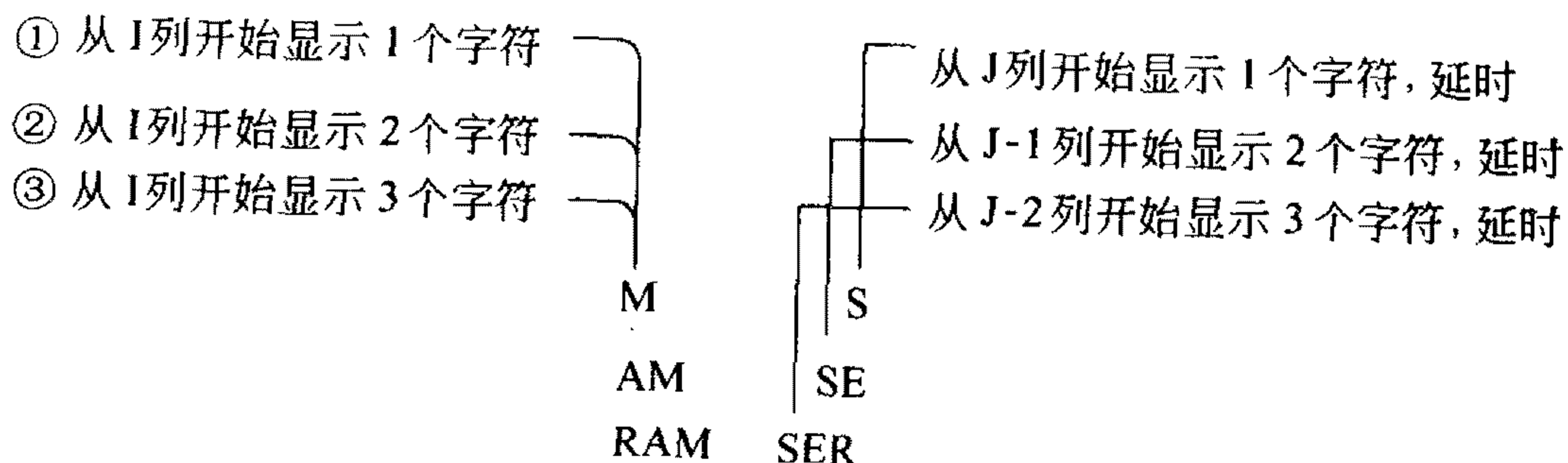


图 5-13 字符串动画显示原理

左移位动画显示的方法:

第1次从J列开始显示1个字符(它是串的首字符),延时 Nms。

第2次从同一行的J-1列开始,显示2个字符(它是串首的2个字符),再延时 Nms。

第3次从同一行的J-2列开始,显示3个字符(它是串首的3个字符),再延时 Nms,以此类推。

右移位动画显示的方法:

第1次从I列开始显示1个字符(它是串的末字符),延时 Nms。

第2次从同一行的I列开始,显示2个字符(它是串的末2个字符),再延时 Nms。

第3次从同一行的I列开始,显示3个字符(它是串的末3个字符),再延时 Nms,以此类推。

左右对进的动画显示,是以上两种方法的交替使用,即在同一行,左移位显示一次,紧接着右移位显示一次,每次各显示1个字符,然后延时 Nms。接下来,仍在同一行左移位显示一次,右移位显示一次,每次各显示2个字符,再延时 Nms……。

只要I值、J值计算正确,当移位次数(左右对进算一次)等于字符串长度时,两串字符就能“重叠”,动画显示到此结束。

我们假设:字符串长度为L,第一次左移位显示的起始列号为J,每次右移位显示的起始列号为I,那么I、J、L一定有如下关系:

$$I = (80 \text{ 列} - L) / 2 \quad J = I + L - 1$$

I、J、L可以用EQU伪指令定义。

【程序清单】

```

;FILENAME: 5111. ASM
.486
DISP    MACRO    Y,X,LENGTH,FLAG,VAR
        MOV      AX,1301H
        MOV      BX,COLOR
        MOV      CX,LENGTH
        MOV      DH,Y
        MOV      DL,X
        IF      FLAG EQ 0
        MOV      BP,OFFSET VAR
        ELSE
        MOV      BP,VAR
        ENDIF
        INT      10H
        ENDM
;-----
DATA    SEGMENT USE16
YYY     EQU      2                ;YYY为上边框行值
XXX     EQU      60              ;XXX为边框长度减2
COLOR   EQU      5FH            ;COLOR为字符串属性字

```

```

TTT      EQU      5                                ;TTT 为延时时间(百分秒)
L1       DB       201,XXX DUP(205),187
L2       DB       186,XXX DUP(' '),186
L3       DB       200,XXX DUP(205),188
LL       EQU      $-L3
XX       EQU      (80-LL)/2
L4       DB       'SERIAL PORTS TEST PROGRAM'
L        EQU      $-L4                                ;L 为串长度
I        EQU      (80-L)/2                            ;I 为右移位显示起始列值
J        EQU      I+L-1                              ;J 为第一次左移位显示
LORG     DB       J+1                                ;的起始列值
COUNT   DW       0                                ;统计移位次数
DATA     ENDS

```

```

;-----
CODE     SEGMENT USE16
        ASSUME   CS: CODE,DS: DATA,ES: DATA
BEG:     MOV     AX,DATA
        MOV     DS,AX
        MOV     ES,AX
        MOV     AX,3
        INT     10H                                ;置彩色文本方式
        DISP    YYY+0,XX,LL,0,L1                    ;显示边框
        DISP    YYY+1,XX,LL,0,L2
        DISP    YYY+2,XX,LL,0,L3
LAST:    INC     COUNT                                ;统计移位次数
        DEC     LORG                                ;列值减 1
        DISP    YYY+1,LORG,COUNT,0,L4                ;左移位显示字符串
        MOV     BP,OFFSET L4+L
        SUB     BP,COUNT
        DISP    YYY+1,I,COUNT,1,BP                    ;右移位显示字符串
        CALL    DELAY                                ;延时
        CMP     COUNT,L                            ;移位次数等于串长度 ?
        JNE     LAST                                ;不等转
        MOV     AH,4CH
        INT     21H                                ;返回 DOS

```

```

;-----
DELAY    PROC                                ;延时 50ms
        MOV     AH,2DH
        MOV     CX,0
        MOV     DX,0
        INT     21H
READ:    MOV     AH,2CH
        INT     21H
        CMP     DL,TTT

```

```

JC      READ
RET
DELAY   ENDP
CODE     ENDS
END      BEG

```

【程序分析】

① 本程序的关键性技巧是设计了一个多参数的显示宏指令 DISP, DISP 宏指令调用了 INT 10H 的 13H 号子功能,定位显示一串彩色字符。它有 5 个哑参数,功能如下:

Y、X 为待显示字符串首字符的行值、列值。

LENGTH 为待显示字符串的长度。

FLAG 为判断标志,若 FLAG 为 0 则 VAR 是存放待显示字符串的变量名,否则 VAR 代表存放待显示字符串的那个单元的偏移地址。由于套用了条件汇编语句进一步增强了 DISP 宏指令的功能。此外 COLOR 单元存放待显示字符串属性字,它的值决定了字符串的颜色。

② 本程序数据段采取了 7 项措施,确保程序的通用性:

改变 YYY 的值,就可以移动标题栏的上、下显示位置。

改变 XXX 的值,就可以加长或缩小标题栏的长度。

改变 COLOR 单元的值,就可以改变标题栏的颜色。

改变 TTT 的值,也就改变了动画的速度。

更换 L4 单元中的字符串,就改变了标题栏的显示内容。

删除“左移位显示字符串”的宏调用,即为右移位动画显示。

删除“右移位显示字符串”的宏调用,即为左移位动画显示。

5.12 模块化程序设计

在设计大型程序的时候,为了加快软件的开发速度,通常采用模块化程序设计的方法,把一个大型程序分解成若干个有相对独立功能的小程序,由多人分别设计,最后链接成一个可执行文件。

什么是“模块”呢? 能够独立汇编的一个逻辑段(例如仅有一个数据段),或者是能够独立汇编的若干个逻辑段的集合,就称为一个模块。每个模块都可以有自己的数据段、附加段、代码段。

在模块化程序中,只允许有一个主模块,其他都是子模块。子模块用“END”作为源程序结束语句,主模块用“END 启动指令标号”作为源程序的结束语句,这是主模块与子模块的识别标志。

既然是模块化程序,模块之间肯定有段间转移、段间调用和段间互访等一系列跨模块的操作,为了使每一个模块都能够独立汇编,汇编语言提供了三条很有用途的伪指令。

5.12.1 支持模块化程序的伪指令

1. PUBLIC 伪指令

格式: PUBLIC 符号名,……,符号名

例如: PUBLIC BEG, NEXT, MESG

功能与说明:

PUBLIC 是公用符号名说明语句,它通知链接程序,语句右侧列出的符号名是本模块中定义的变量名、标号名、过程名,它们要被其他的模块引用。

反之,没有列在 PUBLIC 语句中的符号名,仅能在本模块中使用,而不能被其他模块引用。否则链接时出错。

通常的做法是 PUBLIC 语句放在定义这些符号名的模块的上方。

2. EXTRN 伪指令

格式: EXTRN 符号名: 类型,……,符号名: 类型

例如: EXTRN BEG: FAR, NEXT: FAR, MESG: BYTE

功能与说明:

EXTRN 是外部符号名说明语句,语句右侧的符号名是本模块中引用的,在其他模块中定义过的变量名、标号名或过程名。

符号名右侧的类型,必须是这些符号名在定义时被说明的类型,类型说明符有 BYTE(字节型)、WORD(字型)、DWORD(双字型)、FAR(远)、NEAR(近)。

通常的做法是 EXTRN 语句放在调用模块的上方,它通知汇编程序,所列出的符号名,在其他模块中已经定义过了,否则汇编时出错。

显然,EXTRN 语句应出现在调用模块,而 PUBLIC 语句应出现在被调用模块。

3. INCLUDE 伪指令

格式: INCLUDE 盘符: \路径\文件名. 扩展名

功能: 通知汇编程序把指定的文件复制一份,插入到该语句的下方供汇编时使用。当然,当前盘的盘符、路径可以省略。

5.12.2 模块化程序的设计考虑

① 模块化程序设计之前要有一个总体规划,合理划分模块,使每一个模块有相对独立的功能,尽量减少模块之间的调用。

② 在实地址模式下,链接之后的同类型逻辑段不能超过 64KB。

③ 模块之间,同类型逻辑段的组合与否,是重点考虑的问题。

不同模块中的代码段,可以组合成一个统一的代码段,也可以不组合,仍旧是各自独立的代码段。

不同模块中的数据段(附加段),可以组合成一个统一的数据段(附加段),也可以不

组合,仍旧保持各自的独立性,这些问题应当在总体规划中考虑好。

以代码段为例,怎样把各模块中的代码段组合成一个统一的代码段呢?需要采取以下措施:

- 各模块的代码段选用相同的段名。
- 在代码段定义语句中都选用“PUBLIC”链接参数。

以上措施缺一不可。如果选用相同的分类名更好。

反之,如果不采取以上措施,则链接后各模块的代码段仍旧是各自独立的。

④ 模块之间的转移和调用

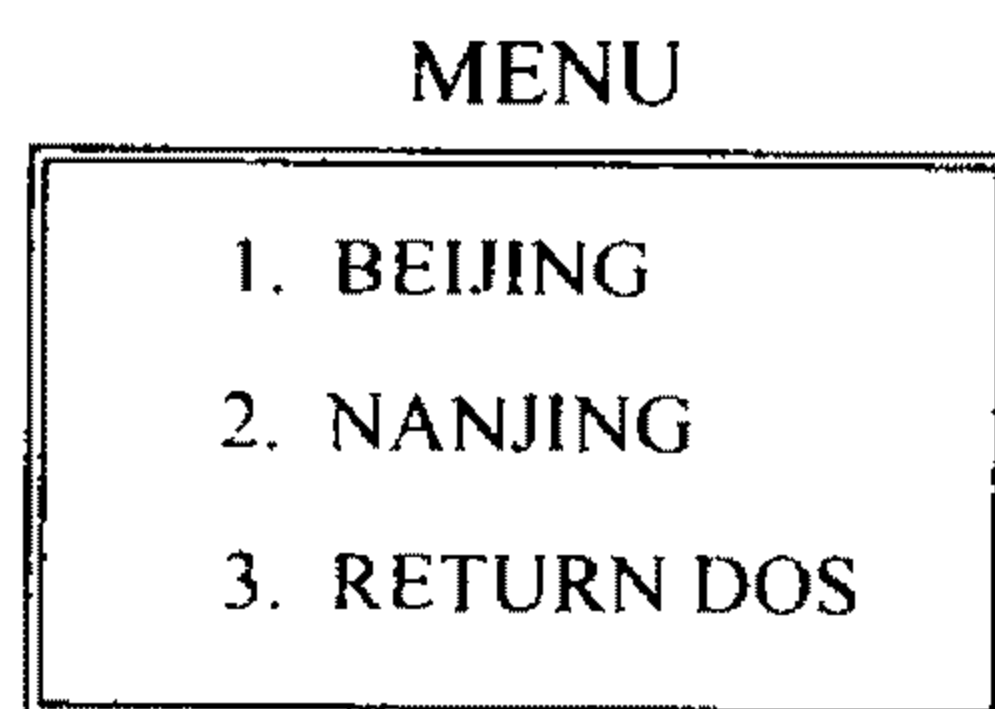
如果各模块之间的代码段,最后要组合成一个代码段,那么模块之间的转移就是段内转移,可以使用段内转移的各种指令格式;模块之间的子程序调用为近过程调用,可以使用段内调用的各种指令格式。反之,如果链接后的代码段仍然是各自独立的,则模块之间转移属于段间转移,模块之间的子程序调用属于远过程调用(被调用的子程序应明确写出有 FAR 属性)。

⑤ 模块之间出现符号名引用的时候必须用 PUBLIC、EXTRN 语句说明。

以上所述是设计模块化程序时必须掌握的基本概念,编程技巧则是次要问题。掌握了模块化程序的设计方法,将使程序员的编程能力更上一个档次。因此模块化程序设计对于高层次程序员颇具诱惑力。

5.12.3 模块化程序设计举例

【例 5.12.1】 设计一个简易的旅游菜单程序,主菜单界面如图 5-14 所示,为了简化程序,突出模块化程序的设计方法,二级菜单不要求介绍相关城市的旅游景点,只给出欢迎词即可。



CHOICE(1,2,3):\$'

图 5-14 旅游主菜单界面

用户输入“1”,转北京二级菜单,屏幕中央显示“WELCOME TO BEIJING!”,底部显示“Please strike any key”,用户敲击任意键之后,程序又转而显示主菜单。

用户输入“2”,转南京二级菜单,屏幕中央显示类似的欢迎词和提示信息……。输入“3”,程序结束返回 DOS。

我们用 3 种方法编程。

【解法 1 设计思路与程序分析】

程序划分为 3 个模块:主模块 5121. ASM 显示主菜单,并完成菜单项的选择和转移,子模块 5121_1. ASM 显示“欢迎到北京来”及提示信息,子模块 5121_2. ASM 显示“欢迎到南京来”及提示信息。

3 个模块都有自己的代码段和数据段,而且模块之间的数据段、代码段不组合,均为各自独立的逻辑段。为此各模块的数据段、代码段均选用不同的段名。模块之间的转移是段间转移,所以在 EXTRN 语句中对于转移标号,均说明为“FAR”。

由于各模块有独立的代码段和独立的数据段,DS 寄存器必须访问自身模块的数据段,所以一旦发生模块之间的转移,DS 的值必须跟踪变化,否则会产生逻辑错误(逻辑错误汇编时发现不了),为此子模块的入口处(BEG1,BEG2)都设置了给 DS 赋值的指令。

【解法 1 程序清单】

```

;FILENAME: 5121. ASM
.486
PUBLIC    BEG
EXTRN     BEG1: FAR,BEG2: FAR
DISP      MACRO    YYXX,VAR
MOV        AH,2           ;光标定位
MOV        BH,0           ;在第 0 页
MOV        DX,YYXX        ;XY 行、XX 列
INT        10H
MOV        AH,9
MOV        DX,OFFSET VAR  ;显示一串字符
INT        21H
ENDM

DATA0     SEGMENT USE16
N         EQU      28 DUP(' ')
L1        DB        N,'          MENU' ,0DH,0AH
          DB        N,'          ' ,0DH,0AH
          DB        N,'          ' ,0DH,0AH
          DB        N,'          ' ,0DH,0AH
          DB        N,'          ' ,0DH,0AH
          DB        N,'          ' ,0DH,0AH
L7        DB        N,'          CHOICE(1,2,3):$ '
DATA0     ENDS
CODE0     SEGMENT 'CODE' USE16
ASSUME     CS: CODE0,DS: DATA0
BEG:      MOV        AX,DATA0
          MOV        DS,AX
          MOV        AX,2
          INT        10H
          DISP        0500H,L1
AGA:      DISP        0B00H,L7
          MOV        AH,1
          INT        21H
          CMP        AL,'3'
          JE         EXIT
          CMP        AL,'1'
          JE         BEG1
          CMP        AL,'2'
          JE         BEG2
          JMP        AGA
EXIT:     MOV        AH,4CH
          INT        21H
CODE0     ENDS

```

1. BEIJING
2. NANJING
3. RETURN DOS

```
END      BEG

;-----
      ;FILENAME: 5121_1. ASM
      .486
      PUBLIC  BEG1
      EXTRN   BEG: FAR
DISP    MACRO  YYXX,VAR
      MOV     AH,2
      MOV     BH,0
      MOV     DX,YYXX
      INT     10H
      MOV     AH,9
      MOV     DX,OFFSET VAR
      INT     21H
      ENDM
DATA1    SEGMENT USE16
MSG1     DB      'WELCOME TO BEIJING ! $ '
ENDMSG   DB      'Please strike any key $ '
DATA1    ENDS
CODE1    SEGMENT 'CODE' USE16
      ASSUME  CS: CODE1,DS: DATA1
BEG1:    MOV     AX,DATA1
      MOV     DS,AX
      MOV     AX,2
      INT     10H
      DISP    0C1EH,MSG1
      DISP    171DH,ENDMSG
      MOV     AH,1
      INT     21H
      JMP     BEG
CODE1    ENDS
      END

;-----
      ;FILENAME: 5121_2. ASM
      .486
      PUBLIC  BEG2
      EXTRN   BEG: FAR
DISP    MACRO  YYXX,VAR
      MOV     AH,2
      MOV     BH,0
      MOV     DX,YYXX
      INT     10H
      MOV     AH,9
      MOV     DX,OFFSET VAR
```




```

                INT      21H
                ENDM
DATA2  SEGMENT  USE16
MSG2   DB      'WELCOME TO NANJING ! $ '
ENDMSG DB      'Please strike any key $ '
DATA2  ENDS
CODE2  SEGMENT  'CODE' USE16
        ASSUME  CS: CODE2,DS: DATA2
BEG2:   MOV     AX,DATA2
        MOV     DS,AX
        MOV     AX,2
        INT     10H
        DISP    0C1EH,MSG2
        DISP    171DH,ENDMSG
        MOV     AH,1
        INT     21H
        JMP     BEG
CODE2   ENDS
        END

```

【解法 2 设计思路与程序分析】

程序也分为 3 个模块：主模块 5122. ASM,子模块 5122_1. ASM 和 5122_2. ASM,它们的功能和解法 1 相同。

解法 2 是另一种思路：把程序中使用的数据集中存放在主模块的数据段中,子模块不设置数据段,另外,3 个模块的代码段,最后要组合成一个统一的代码段。为此 3 个模块的代码段都选用了相同的段名,都选用“PUBLIC”做为链接参数。

由于要组合成一个统一的代码段,所以模块之间的转移是段内转移,EXTRN 语句中,对转移标号用 NEAR 说明它们是近距离转移。

有一种情况是不可避免的：某一个标号既是段内转移的目标,又是段间转移的目标,在这种情况下 EXTRN 语句对于这些标号名、过程名,应当写成“FAR”。

由于 3 个模块使用一个数据段,因此在主模块完成对 DS 赋值之后,转入其他模块时不应破坏 DS 的初值。

【解法 2 程序清单】

```

;FILENAME: 5122. ASM
.486
PUBLIC      BEG,MSG1,MSG2,ENDMSG
EXTRN       BEG1: NEAR,BEG2: NEAR
DISP        MACRO YYXX,VAR
MOV         AH,2
MOV         BH,0
MOV         DX,YYXX
INT         10H

```

```

MOV      AH,9
MOV      DX,OFFSET VAR
INT      21H
ENDM

DATA     SEGMENT  USE16
N        EQU      28 DUP(' ')
L1       DB      N,'          MENU          ',0DH,0AH
         DB      N,'          ',0DH,0AH
         DB      N,'          1. BEIJING      ',0DH,0AH
         DB      N,'          2. NANJING     ',0DH,0AH
         DB      N,'          3. RETURN DOS  ',0DH,0AH
         DB      N,'          ',0DH,0AH
L7       DB      N,'          CHOICE(1,2,3):$ '
MSG1     DB      'WELCOME TO BEIJING ! $ '
MSG2     DB      'WELCOME TO NANJING ! $ '
ENDMSG   DB      'please strike any key $ '
DATA     ENDS

CODE     SEGMENT  BYTE PUBLIC 'CODE' USE16
ASSUME   CS: CODE,DS: DATA

BEG:     MOV      AX,DATA
         MOV      DS,AX
         MOV      AX,2
         INT      10H
         DISP     0500H,L1
AGA:     DISP     0B00H,L7
         MOV      AH,1
         INT      21H
         CMP      AL,'3'
         JE       EXIT
         CMP      AL,'1'
         JE       BEG1
         CMP      AL,'2'
         JE       BEG2
         JMP      AGA
EXIT:    MOV      AH,4CH
         INT      21H
CODE     ENDS
END      BEG
;-----
;FILENAME: 5122_1. ASM
.486
PUBLIC  BEG1
EXTRN  MSG1: BYTE,NEXT: NEAR
DISP   MACRO  YYXX,VAR

```



```

MOV      AH,2
MOV      BH,0
MOV      DX,YYXX
INT      10H
MOV      AH,9
MOV      DX,OFFSET VAR
INT      21H
ENDM
CODE     SEGMENT  BYTE PUBLIC 'CODE' USE16
        ASSUME   CS: CODE
BEG1:    MOV      AX,2
        INT      10H
        DISP     0C1EH,MESG1
        JMP      NEXT
CODE     ENDS
END

;-----
;FILENAME: 5122_2. ASM
.486
PUBLIC   BEG2,NEXT
EXTRN    MSG2: BYTE,ENDMSG: BYTE,BEG: NEAR
DISP     MACRO   YYXX,VAR
MOV      AH,2
MOV      BH,0
MOV      DX,YYXX
INT      10H
MOV      AH,9
MOV      DX,OFFSET VAR
INT      21H
ENDM
CODE     SEGMENT  BYTE PUBLIC 'CODE' USE16
        ASSUME   CS: CODE
BEG2:    MOV      AX,2
        INT      10H
        DISP     0C1EH,MESG2
NEXT:    DISP     171DH,ENDMSG
        MOV      AH,1
        INT      21H
        JMP      BEG
CODE     ENDS
END

```

5.12.4 宏指令共享

以上两种解法,每一个模块中都定义了一条宏指令 DISP,该指令完成光标定位与字

符串显示,宏指令的名称、结构和参数都相同。能不能只定义一次,各模块共享呢?可以!汇编语言为此设计了一条 INCLUDE 伪指令。怎样使用 INCLUDE 伪指令实现宏指令共享的目的?需要预先做一项准备工作:把要共享的宏指令单独组织一个文件(注意:只编辑不汇编),存放在指定的目录下,然后在调用模块中用 INCLUDE 伪指令说明即可。本例将 DISP 宏指令单独组织一个文件,文件名为“DISPLAY. MAC”,该文件和主模块文件以及两个子模块文件都放在同一个目录下,因此,INCLUDE 伪指令可以不必写出该文件的盘符和路径。

我们以解法 2 为基础,运用宏指令共享技术,设计出较为满意的第 3 种解法的程序,该程序由 4 个源文件组成。

【解法 3 程序清单】

```

;FILENAME: DISPLAY. MAC
DISP    MACRO    YYXX,VAR
        MOV      AH,2
        MOV      BH,0
        MOV      DX,YYXX
        INT      10H
        MOV      AH,9
        MOV      DX,OFFSET VAR
        INT      21H
        ENDM
;-----
;FILENAME: 5123. ASM
.486
PUBLIC  BEG,MESG1,MESG2,ENDMESG
EXTRN  BEG1: NEAR,BEG2: NEAR
INCLUDE DISPLAY. MAC
DATA    SEGMENT USE16
N        EQU      28 DUP(' ')
L1       DB        N,'          MENU          ',0DH,0AH
        DB        N,'          ',0DH,0AH
        DB        N,'          1. BEIJING      ',0DH,0AH
        DB        N,'          2. NANJING      ',0DH,0AH
        DB        N,'          3. RETURN DOS    ',0DH,0AH
        DB        N,'          ',0DH,0AH
L7       DB        N,'          CHOICE(1,2,3):$ '
MESG1    DB        'WELCOME TO BEIJING ! $ '
MESG2    DB        'WELCOME TO NANJING ! $ '
ENDMESG  DB        'please strike any key $ '
DATA     ENDS
CODE     SEGMENT BYTE PUBLIC 'CODE' USE16
        ASSUME    CS: CODE,DS: DATA
BEG:     MOV      AX,DATA

```



```

                MOV     DS,AX
                MOV     AX,2
                INT     10H
                DISP    0500H,L1
AGA:            DISP    0B00H,L7
                MOV     AH,1
                INT     21H
                CMP     AL,'3'
                JE      EXIT
                CMP     AL,'1'
                JE      BEG1
                CMP     AL,'2'
                JE      BEG2
                JMP     AGA
EXIT:           MOV     AH,4CH
                INT     21H
CODE            ENDS
                END      BEG

```

```

;FILENAME: 5123_1. ASM
.486
PUBLIC         BEG1
EXTRN          MSG1: BYTE,NEXT: NEAR
INCLUDE        DISPLAY. MAC
CODE            SEGMENT BYTE PUBLIC 'CODE' USE16
                ASSUME  CS: CODE
BEG1:           MOV     AX,2
                INT     10H
                DISP    0C1EH,MSG1
                JMP     NEXT
CODE            ENDS
                END

```

```

;FILENAME: 5123_2. ASM
.486
PUBLIC         BEG2,NEXT
EXTRN          MSG2: BYTE,ENDMSG: BYTE,BEG: NEAR
INCLUDE        DISPLAY. MAC
CODE            SEGMENT BYTE PUBLIC 'CODE' USE16
                ASSUME  CS: CODE
BEG2:           MOV     AX,2
                INT     10H
                DISP    0C1EH,MSG2
NEXT:           DISP    171DH,ENDMSG

```

```
MOV      AH,1
INT      21H
JMP      BEG
CODE     ENDS
END
```

【小结】

模块化程序可执行文件的生成步骤如下：

① 分别对各个模块进行编辑和汇编,生成各自的目标文件(共享的宏文件只编辑不汇编),假设解法3的目标文件为:5123.OBJ,5123_1.OBJ,5123_2.OBJ。

② 然后用链接程序进行链接:例如:

```
TLINK 5123+5123_1+5123_2
```

生成的可执行文件与第一个目标文件同名,本例的可执行文件为5123.EXE。

习 题

1. 数据段 NUMBER 单元有一个数 X,判断 $5 < X \leq 24$? 若是置 FLAG 单元为 0,否则置 FLAG 单元为 -1。

2. 计算 $1+2+3+\dots+199+200$,和数→SUM 单元。

3. 设数据段 BUF 单元开始有 10 个有符号的单字节数,其中必定有负数,找出其中真值最小的数→屏幕显示。

显示格式为 MIN= -×××××××B

4. 由键盘输入任意两位十进制数,然后转换成一字节 BCD 码→数据段 BCD 单元。

5. 由键盘输入任意组合的八个 0、1 字符,然后转换成等值的二进制数送数据段 BEN 开始的字节型单元。

6. 由键盘输入两个 3 位十进制数(一个 3 位十进制数以回车做为结束标志),转换成等值的二进制数→数据段两个字型单元。

7. 由键盘输入两位十六进制数,然后转换成等值的十进制数→屏幕显示。

8. 设计一条宏指令,它的功能是为宏参数 RM 中的标准 ASCII 码设置偶校验。注意:该指令可能被多次调用,调用时实参数可以是寄存器操作数或者内存操作数。

9. 对物理地址为 B8000H 开始的 4000 个单元进行写操作,奇地址单元写入 1FH,偶地址单元写入 41H。用传送指令(配合循环)和串操作指令分别设计一个程序,并预测程序的执行结果。

第 6 章

chapter 6

总线

6.1 总线基本概念

总线是构成计算机系统的互连机构,是多个系统功能部件之间进行数据传送的公共通路,通过总线可以传输数据信息、地址信息、各种控制命令和状态信息。

在计算机的发展历史中,早期冯·诺依曼提出的模型并不包含总线,到微型计算机技术发展以后,才正式采用总线结构。有了总线结构以后,计算机系统的组装、维护和扩展才得以方便地进行,使系统具有了支持模块化设计、开放性、通用性和灵活性等特点。

6.1.1 总线的类型与总线结构

1. 总线的类型

一个系统常常包含了多种类型的总线。计算机系统的总线按其所传输信号的性质分为三类:地址总线、数据总线和控制总线。地址总线和数据总线相对比较简单,功能也较为单一。尽管在系统的不同层面上它们的名称和性能有所不同,但地址总线和数据总线的功能就是传输、交换地址信息和数据信息。控制总线差异较大,这一特点决定了各种模块的不同接口和功能特点。

整个计算机系统包含有许多模块,这些模块位于系统的不同层次上,整个系统按模块进行构建。同一类型的总线在不同的层面上连接不同部位上的模块,其名称、作用、数量、电气特性和形态各不相同。按总线连接的对象和所处系统的层次来分,总线有芯片级总线、系统总线、局部总线和外部总线。芯片级总线用于模块内芯片级的互连,是该芯片与外围支撑芯片的连接线。如连接 CPU 及其周边的协处理器、总线控制器、总线收发器等总线称为 CPU 局部总线或 CPU 总线,连接存储器及其支撑芯片的总线称为存储器总线。系统总线是连接计算机内部各模块的一条主干线,是连接芯片级总线、局部总线和外部总线的纽带。系统总线符合某一总线标准,具有通用性,是计算机系统模块化的基础。由于经过缓冲器驱动,负载能力较强。与所连接的 CPU 和外部设备相比,系统总线发展滞后、速度缓慢、带宽较窄,成为数据传输的瓶颈。为了打破这一瓶颈,人们将一些高速外设从系统总线上卸下,通过控制和驱动电路直接挂到 CPU 局部总线上,使高速外设能按 CPU 速度运行。这种直接连接 CPU 和高速外围设备的传输通道就是局部

总线。局部总线一端与 CPU 连接,另一端与高速外设和系统总线连接,好像在系统总线和 CPU 总线之间又插入一级。外部总线又称设备总线或输入输出总线,是连接计算机与外部设备的总线。外部总线经总线控制器挂接到系统总线上。CPU 与连接到系统板上的外设打交道须经过芯片级总线、局部总线、(系统总线)和外部总线这样三到四级总线。

总线按照允许信息传送的方向来分,可分为单向传输和双向传输两种。双向传输又分为半双向和全双向两种。前者在同一时刻只允许向其中的一个方向进行数据传送,而在另一时刻可以实现反方向的数据传送。后者允许在同一时刻进行两个方向的数据传送。全双向的总线速度快,但造价高,结构复杂。

按照用法,总线又可以分为专用总线和非专用总线。只连接一对物理部件的总线称为专用总线。其基本优点是系统的流量高,多个部件可以同时发送和接收信息,几乎不争用总线。控制简单,不用指明源和目标。任何总线的失效只会影响连接到该总线的两个部件不能直接通信,但它们仍可通过其他部件间接通信,因而系统可靠。专用总线的主要缺点是总线数目多;难以小型化和集成电路化,而且总线长时,成本高。另外,专用总线的时间利用率往往很低,系统的模块化也较难实现。专用总线只适用于实现某个设备(部件)与另一个设备(部件)的相连。非专用总线可以被多种功能或多个部件分时共享,同一时间只有一对部件可以使用总线进行通信。非专用总线的主要优点是总线数少,造价低;总线接口的标准化、模块性强,易于简化和统一接口的设计;可扩充能力强,部件的增加不会使电缆、接口和驱动电路等剧增;易于采用多重总线来提高总线的带宽和可靠性,使故障弱化。缺点是系统流量小,经常会出现总线争用,使那些未获得总线使用权的部件不得不等待而降低效率。如果处理不当,总线可能成为系统速度性能的瓶颈,对单总线结构来说更是如此。

2. 总线结构

总线用来连接系统内各模块,组织方法不同,总线结构也不同。一般的有单总线结构、双总线结构和三总线结构。

单总线结构是指在许多单处理器的计算机中,使用一条单一的系统总线来连接 CPU、主存和 I/O 设备。此时要求连接到总线上的逻辑部件必须高速运行,以便在某些设备需要使用总线时能迅速获得总线控制权;而当不再使用总线时,能迅速放弃总线控制权。单总线结构容易扩展成多 CPU 系统,只需要在系统总线上挂接多个 CPU 即可。

双总线结构保持了单总线系统简单、易于扩充的优点,但又在 CPU 和主存之间专门设置了一组高速的存储总线,使 CPU 可通过专用总线与存储器交换信息,并减轻了系统总线的负担,同时主存仍可通过系统总线,与外设之间实现 DMA 操作,而不必经过 CPU。当然这种双总线系统以增加硬件为代价。三总线结构是在双总线系统的基础上增加 I/O 总线形成的。

6.1.2 总线的性能

总线的性能主要从以下三方面来衡量:总线宽度、总线频率和传输率。

1. 总线宽度

总线宽度是指一次可以同时传输的数据位数。一般来说,总线的宽度越宽,在一定时间内传输的信息量就越大。不过在一个系统中,总线的宽度不会超过 CPU 的数据宽度。

2. 总线频率

总线频率是指总线工作时每秒内能传输数据的次数。总线的频率越高,传输的速度越快。

3. 传输率

传输率是指每秒内能传输的字节数,用 MB/s 来表示。

传输率和宽度、频率之间的关系是:

$$\text{传输率} = \text{宽度} / 8 \times \text{频率}$$

【例 6.1.1】 设总线宽度为 32 位,频率为 100MHz,求传输率。

根据上述公式,

$$\text{传输率} = 32\text{b} / 8 \times 100\text{MHz} = 400\text{MB/s}$$

又如,PCI 总线的宽度为 32 位,总线频率为 33MHz,所以,PCI 的数据传输率为 132MB/s。总线宽度越宽,频率越高,则传输率越高。

6.1.3 总线信息的传送方式

数字计算机使用二进制数,它们或用电位的高、低来表示,或用脉冲的有、无来表示。计算机系统中,总线传输方式即总线通信方式,俗称总线握手方式。总线上信息的传送方式一般有三种:串行传送、并行传送和分时传送。出于速度和效率上的考虑,系统总线上传送的信息必须采用并行传送方式。

1. 串行传送

当信息以串行方式传送时,只有一条传输线,且采用脉冲传送。在串行传送时,按顺序传送来表示一个数码的所有二进制位(bit)的脉冲信号,每次一位。通常以第一个脉冲信号表示数码的最低有效位,最后一个脉冲信号表示数码的最高有效位。进行串行传送时,被传送的数据需要在发送部件进行并-串变换,而在接收部件又需要进行串-并变换。串行传送的主要优点是只需要一条传输线,这一点对长距离传输显得特别重要,不管传送的数据量有多少,只需要一条传输线,成本比较低廉。

2. 并行传送

用并行方式传送二进制信息时,对要传送的数码的每个数据位都需要一条单独的传输线。信息由多少二进制位组成,就需要有多少条传输线,从而使得二进制数“0”或“1”在不同的线上同时进行传送。

并行传送一般采用电位传送。由于所有的位同时被传送,所以并行数据传送比串行数据传送快得多。

3. 分时传送

分时传送有两种概念。一种是采用总线复用方式,某个传输线上既传送地址信息,又传送数据信息。为此必须划分时间片,以便在不同的时间间隔中完成地址传送和数据传送的任务。分时传送的另一种概念是共享总线的部件,分时使用总线。

6.2 典型总线标准

相同的指令系统,相同的功能,不同厂家生产的各功能部件在实现方法上几乎没有相同的,但各厂家生产的相同功能部件却可以互换使用,其原因在于它们都遵守了相同的总线的要求,这就是总线的标准化问题。

总线标准往往由相关生产厂家首先提出,他们对连接总线的插件的几何尺寸、引脚排序、电路信号名称及其电气特性等都作了详细的规定,成为实际的工业标准,然后获得行业或国际标准组织的批准,即成为大家接受的某种总线标准。

目前总线接口的标准化已经做到在系列机内不同档次的机器都采用统一的 I/O 总线规范。20 世纪 70 年代中期的 S-100 总线曾广泛用于微、小型机中,20 世纪 70 年代末期的 IEEE-488 标准 I/O 总线也得到广泛采用。20 世纪 80 年代以来,各计算机厂家与有关的 IEEE 标准委员会合作开发了大量的底版总线标准。1991 年推出的 Futurebus+ 总线标准是迄今为止最复杂的总线标准,能支持 64 位地址空间,64 位、128 位和 256 位数据传输,为下一代的多处理机系统提供了一个稳定的平台。它可以满足各类高性能系统的需求,因此适合于高成本的较大规模计算机系统。PCI 总线是当前最流行的总线,是一个高带宽且与处理器无关的标准总线,又是至关重要的层次总线。它采用同步定时协议和集中式仲裁策略,并具有自动配置能力。PCI 与其他不同的总线规范相比,为高速的 I/O 子系统(例如图形显示适配器、网络接口控制器和磁盘控制器等)提供了更好的性能。适合于低成本的小系统,因此在微型机系统中得到了广泛的应用。

本节就微型计算机系统中常用的几类总线标准作出阐述。

6.2.1 AT 总线

以 80286 作为微处理器的 PC/AT 机,使用 AT 总线(ISA 总线)。AT 总线的数据宽度为 16 位,工作频率为 8MHz,数据传输率最高为 8MB/s。

AT 总线在 PC 总线(62 芯插槽)的基础上增加了一个 36 芯的副插槽,使数据线增加到 16 根,地址线增加到 24 根,可以寻址 16 兆地址空间。也就是说,AT 总线的主插槽与 PC 总线是兼容的。IBM 发布了 AT 总线标准后,许多厂家纷纷生产兼容机,使 PC 系列微型机销售量占了整个微型计算机市场的 70%,因此 AT 总线标准也就成为事实上的工业总线标准了,所以 AT 总线又称 ISA(Industry Standard Architecture)总线。

386SX 档次以下的兼容机大都采用 AT 总线结构。

AT 总线插槽引脚分配如图 6-1 所示。

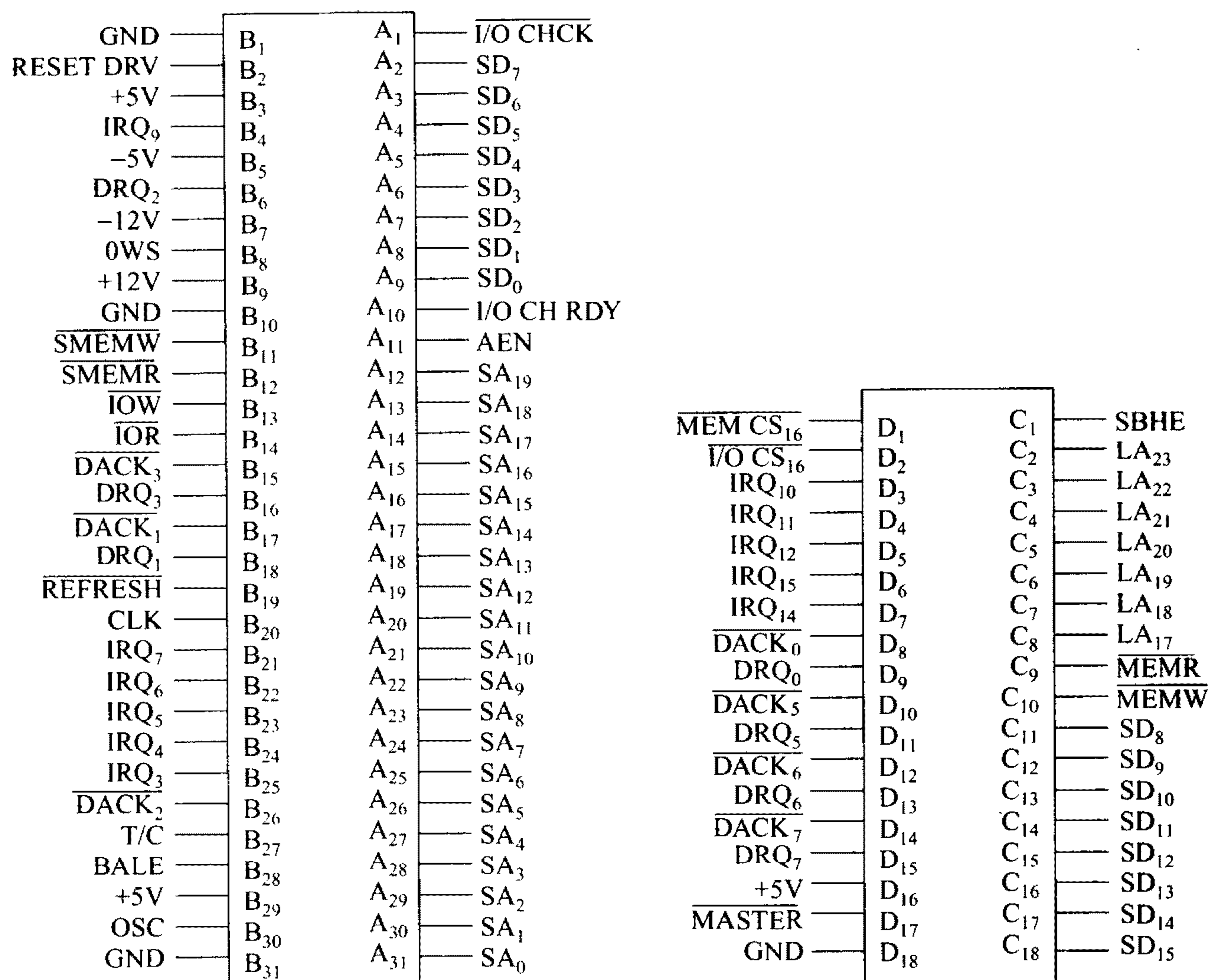


图 6-1 AT 总线插槽引脚分配

AT 总线 62 芯插槽引脚的信号分布与功能定义与 PC 总线基本相同,分为五类:

(1) 时钟与复位

OSC: 周期为 70ns 的振荡信号。

CLK: 频率为 6MHz,周期为 167ns 的系统时钟。

RESET DRV: 复位信号。

0WS: 零等待状态输入,通知 CPU 无须插入等待周期即可完成当前的总线操作。

(2) 数据线

SD₇~SD₀: 8 根双向数据线。

(3) 地址线

SA₁₉~SA₀: 20 根地址线,提供对存储器和 I/O 端口寻址。

(4) 控制线

BALE: 由 82288 总线控制器提供,允许锁存来自 CPU 的有效地址。

AEN: 禁止 CPU 和其他 I/O 端口使用系统总线,允许 DMA 控制器控制地址总线、数据总线和读/写命令线,进行 DMA 传送。

IRQ₉、IRQ₃~IRQ₇: I/O 端口的中断请求线,IRQ₉ 优先级最高(由于 AT 机中 IRQ₂

用作从片 8259 的中断申请,因此 IRQ_2 不出现在 62 芯插槽引脚中), IRQ_9 (B4 端子)即是用户的中断请求输入端。

$DRQ_1 \sim DRQ_3$: I/O 端口的 DMA 请求线。

$\overline{DACK}_1 \sim \overline{DACK}_3$: DMA 应答信号线。

T/C: 当任一 DMAC 通道计数结束时,由 DMA 控制器发出。

\overline{IOR} : I/O 端口读命令。

\overline{IOW} : I/O 端口写命令。

\overline{SMEMR} : 存储器读命令,仅对低于 1MB 的存储空间有效。

\overline{SMEMW} : 存储器写命令。仅对低于 1MB 的存储空间有效。

$\overline{I/O\ CH\ CK}$: I/O 通道奇偶校验出错信号。

$\overline{I/O\ CH\ RDY}$: I/O 通道准备好信号。

$\overline{REFRESH}$: 用以表明刷新周期。

(5) 电源与地线

+5V, -5V, +12V, -12V, GND

AT 总线 36 芯插槽引脚分四类:

(1) 数据线

$SD_{15} \sim SD_8$: 双向数据线,和 $SD_7 \sim SD_0$ 一起构成 16 位数据总线。

(2) 地址线

$LA_{23} \sim LA_{17}$: 非锁存地址线,使系统有 16 兆字节的寻址能力,这些信号在 ALE 为高电平时有效。

(3) 控制线

SBHE: 数据高位允许信号。

$\overline{MEM\ CS}_{16}$: 该信号有效表示当前的数据传送周期,是具有 1 个等待状态的 16 位存储器读写周期。

$\overline{I/O\ CS}_{16}$: 该信号有效表示当前的数据传送周期,是具有 1 个等待状态的 16 位 I/O 端口读写周期。

$IRQ_{10} \sim IRQ_{12}, IRQ_{14} \sim IRQ_{15}$: 中断请求信号。

$DRQ_0, DRQ_5 \sim DRQ_7$: DMA 请求信号, DRQ_0 为最高级。

$\overline{DACK}_0, \overline{DACK}_5 \sim \overline{DACK}_7$: DMA 应答信号。

MASTER: 输入信号,它和 DRQ 共同作用,使 I/O 通道上的 DMA 控制器获得对系统总线的控制。

\overline{MEMR} : 存储器读命令(对所有的存储空间执行存储器读周期时有效)。

\overline{MEMW} : 存储器写命令(对所有的存储空间执行存储器写周期时有效)。

(4) 电源和地线

+5V, GND

80386 芯片问世后,由于 CPU 是 32 位,如果仍采用 ISA 总线,将使 CPU 性能不能充分发挥。因此 IBM 公司在推出它的第一台 386 微型计算机时,开发了与 ISA 标准完全不同的系统总线标准: MCA 总线(即“微通道”)。由于 IBM 对 MCA 总线实行保护政

策,致使以 Compaq 为代表的美国九大计算机厂家联合制定了一种 32 位总线结构——EISA (Extended Industrial Standard Architecture)总线。该总线是 ISA 总线的 32 位扩展,与 ISA 总线兼容,它具有 32 位数据线,33MB/s 的数据传输率,提供多处理器控制功能(multi-master),其多主控总线使一般计算机的单处理器环境升级至多处理器环境,而且扩展卡安装方便,自动配置,无须跳线,保持了与 ISA 总线百分之百的兼容。

EISA 总线将 ISA 总线的扩展插座加深,底部又增加了一层插芯,形成上下两层,总共近 200 个信号。上层信号完全与 ISA 总线一样,下层增加 EISA 专用信号。由于下层定位匙(Access Key)的作用,ISA 卡在 EISA 总线插座上插不到底,仅与上层插芯接触,只用 ISA 总线信号;而 EISA 卡在本系统总线插座上,对应下层定位匙处都开出定位槽,可以插到底部,使深层的插芯也能可靠接触,于是使用全部 EISA 总线信号。这种方式实现了两种总线标准插卡的兼容。

EISA 总线的主要特点有:

① 将数据总线扩展到 32 位($D_{15} \sim D_0$ 在上层插芯, $D_{31} \sim D_{16}$ 在下层插芯,仅 EISA 卡可接触)。4 字节的数据总线用控制总线信号 $\overline{EX32}$ 、 $\overline{EX16}$ 、 \overline{SBHE} 、 $\overline{BE_3} \sim \overline{BE_0}$ 分别进行控制,实现双字、字和字节的传送。

② 支持突发传送(Burst Transfer)。突发传送又称猝发传送、成组传送,是指传送存储器中连续存放的一组数据(字或双字)时,第一个同步时钟 BCLK 周期往总线上发送首地址,第二个 BCLK 周期即传送第一个数;以后地址自动增量,不再占用 BCLK 周期发送地址,后续每个数只需一个 BCLK 周期即可完成传送。所以在 BCLK 频率为 8.333MHz 时,以双字为单位传输的最高速率可达 33MB/s。

③ 地址总线扩展到 32 位,可直接寻址 4GB 的物理空间。

6.2.2 PCI 总线

系统总线虽然从 PC 总线、ISA 总线发展到 EISA 总线,但仍然跟不上软件和 CPU 的发展速度,在大部分时间内,CPU 仍处于等待状态。

随着 CPU 芯片不断更新换代和各种高速适配卡的出现,加上操作系统及应用程序越来越复杂,ISA/EISA 总线已满足不了快速的数据传输,可以说是总线,而不是外设妨碍了系统整体性能的提高,这是单一慢速的系统总线结构带来的限制。

1991 年,出现了局部总线标准 VESA,比 EISA 性能更完善,传输速率更高,它将外设直接挂接到 CPU 局部总线上,并以 CPU 的速度运行,极大地提高了外设的运行速度。

VESA 总线的数据宽度为 32 位,可以扩展到 64 位,与 CPU 同步工作,最大运行速度可达 66MHz,VESA 的最大传输率达到 132MB/s,是 ISA 总线传输率的 16 倍。

但是,VESA 总线存在规范定义不严格,兼容性差,总线速度受 CPU 速度影响等缺陷。比 VESA 更强的是 1992 年推出的局部总线——PCI (Peripheral Component Interconnect)总线。PCI 比 VESA 规范定义严格,因而保证了良好的兼容性。PCI 总线主要是为奔腾微处理器的开发使用而设计的,也支持 80386/80486 微处理器系统。

随着高档微型计算机的发展,为了与早期的微型计算机系统兼容,如今微型计算机

系统结构多采用不同总线构成的多总线结构,在主机板上留有不同总线的插槽。目前国内微型计算机市场上,486 微型计算机使用 ISA 总线(AT 总线)和 VESA 总线,586 以上微型计算机使用 ISA 总线(AT 总线)和 PCI 总线。

PCI 总线结构中的关键部件是 PCI 总线控制器,这是一个复杂的管理部件,用来协调 CPU 与各种外设之间的数据传输,并提供统一的接口信号。

1. PCI 总线特点

PCI 总线的主要特点如下:

(1) 传输率

PCI 用 32 位数据传输,也可扩展为 64 位。用 32 位数据宽度时,以 33MHz 的频率运行,传输率达 132MB/s,用 64 位数据宽度时,以 66MHz 的频率运行,传输率达 528MB/s。PCI 的高传输率为多媒体传输和高速网络传输提供了良好的支持。

(2) 高效率

PCI 总线控制器中集成了高速缓冲器,当 CPU 要访问 PCI 总线上的设备时,可把一批数据快速写入 PCI 缓冲器,此后,PCI 缓冲器中的数据写入外设时,CPU 可执行其他操作,从而使外设和 CPU 并发运行,所以效率得到很大提高。此外,PCI 总线控制器支持突发数据传输模式,用这种模式,可以实现从一个地址开始通过地址加 1 连续快速地传输大量数据,减少了地址译码环节,从而有效利用总线的传输率,这个功能特别有利于高分辨率彩色图像的快速显示以及多媒体传输。

(3) 即插即用功能

即插即用功能是由系统和适配器两个方面配合实现的。在适配器角度,为了实现即插即用功能,制造商都要在适配器中增加一个小型存储器存放按照 PCI 规范建立的配置信息。配置信息中包括制造商标识码、设备标识码以及适配器的分类码等,含有向 PCI 总线控制器申请建立配置表所需要的各种参数,如存储空间的大小、I/O 地址和中断源等。在系统角度,PCI 总线控制器能够自动测试和调用配置信息中的各种参数,并为每个 PCI 设备配置 256 字节的空间来存放配置信息,支持其即插即用的功能。在系统加电时,PCI 总线控制器通过读取适配器中的配置信息,为每个卡建立配置表,并对系统中的多个适配器进行资源分配和调度,实现即插即用功能。在添加新的扩展卡时,PCI 控制器能够通过配置软件自动选用空闲的中断号,确保 PCI 总线上的各扩展卡不会冲突,从而为新的扩展卡提供即插即用环境。

(4) 独立于 CPU

PCI 控制器用独特的与 CPU 结构无关的中间连接件机制设计,这一方面使 CPU 不再需要对外设直接控制,另一方面由于 PCI 总线机制完全独立于 CPU,从而支持当前的和未来的各种 CPU,能够在未来有长久的生命期。

(5) 负载能力强、易于扩展

PCI 的负载能力比较强,而且 PCI 总线上还可以连接 PCI 控制器,从而形成多级 PCI 总线,每级 PCI 总线可以连接多个设备。

(6) 兼容各类总线

PCI 总线设计考虑了和其他总线的配合使用,能够通过各种“桥”兼容和连接以往的

多种总线。所以在 PCI 总线系统中,往往还有其他总线存在。

PCI 总线控制器像桥梁一样一边连接 CPU 总线,另一边连接 CPU 访问相对频繁、速度也较快的部件,因此,PCI 总线控制器也被称为“PCI 桥”。PCI 桥也可称为 PCI 桥接器,事实上是一个总线转换部件,其功能是连接两条计算机总线,允许总线之间相互通信交往。一座桥的主要作用是把一条总线的地址空间映射到另一条总线的地址空间,就可以使系统中每一个主设备(Master)能看到同样的一份地址表。这时,从整个存储系统来看,有了整体性统一的直接地址表,可以大大简化编程模型。桥本身可以十分简单,比如只是加上信号的缓冲能力;也可以相当复杂,比如可以包括有组织转换数据及快存化,装拆数据分组以及由各类系统所规定的一些功能。

PCI 规范中提出了三类桥的设计:主 CPU 至 PCI 的桥(主桥);PCI 至标准总线(如 ISA、EISA)之间的“标准总线桥”;PCI 至 PCI 之间的桥。

2. PCI 总线信号

在一个 PCI 应用系统中,有主设备和从设备。如果只作为从设备,则至少需要 47 根信号线,若作为主设备则需要 49 根信号线。利用这些信号线可以处理数据和地址,实现接口控制、仲裁和系统功能。PCI 总线信号如图 6-2 所示,图中左边为必要信号,右边为任选信号。

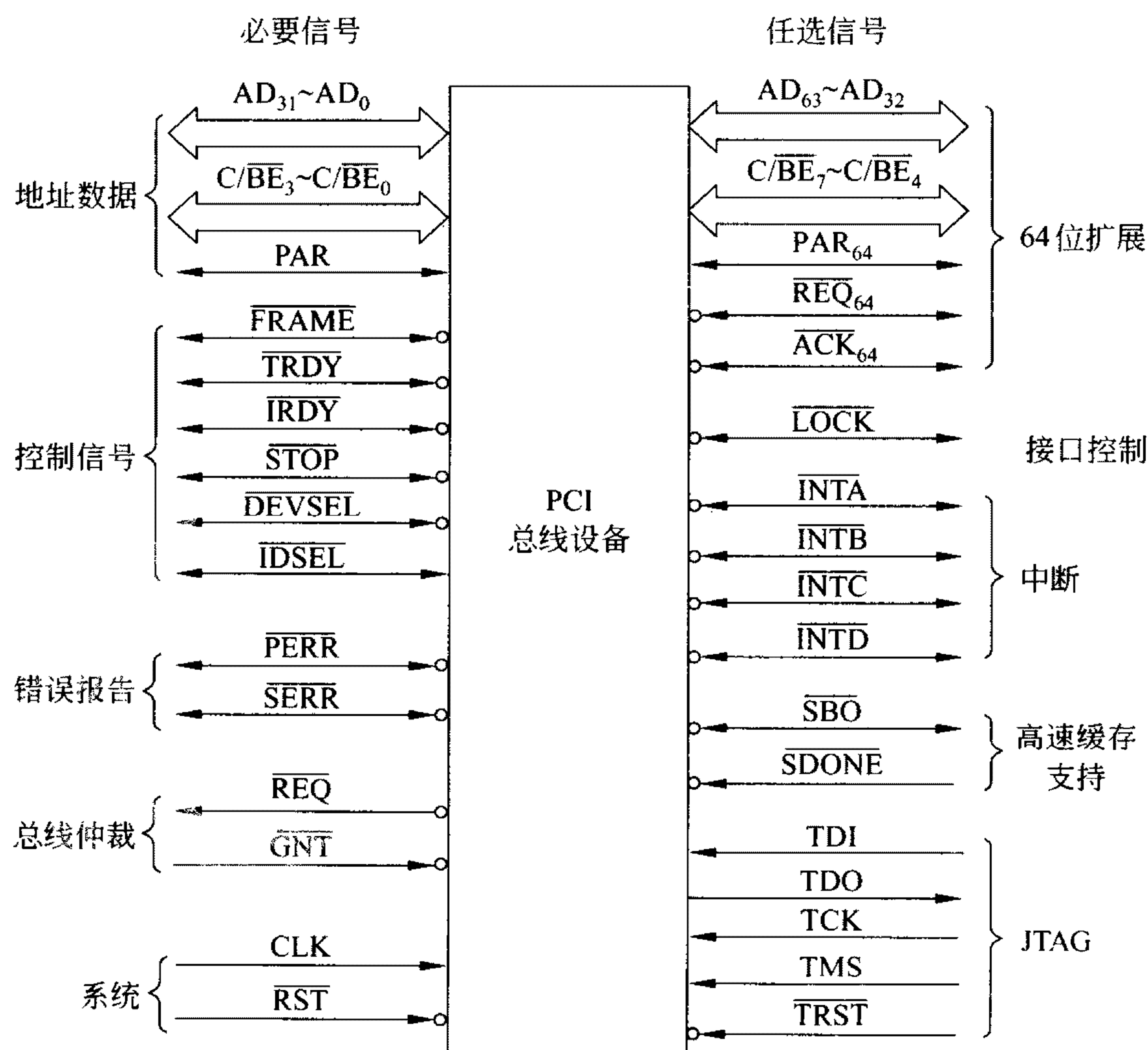


图 6-2 PCI 总线信号

从图可知,PCI 总线可按功能分为几组,分别是:

(1) 地址数据信号

$AD_{31} \sim AD_0$ ：32 位地址数据复用信号。在 PCI 总线传输时,包含一个地址传送节拍和一个(或多个)数据传送节拍,在 \overline{FRAME} (帧周期信号)有效时为地址传送节拍开始,在 \overline{IRDY} (主设备就绪信号)和 \overline{TRDY} (从设备就绪信号)同时有效时为数据传送节拍。

$C/\overline{BE}_3 \sim C/\overline{BE}_0$ ：总线命令/字节允许信号。在地址传送节拍传送 PCI 总线命令,在数据传送节拍传送字节允许信号, C/\overline{BE}_0 对应字节为 0。

总线命令由主机发向从设备,说明当前事务类型,总线命令在地址节拍呈现在 $C/\overline{BE}_3 \sim C/\overline{BE}_0$ 上并被译码。PCI 总线命令说明如表 6-1 所示。

表 6-1 PCI 总线命令

$C/\overline{BE}_3 \sim C/\overline{BE}_0$	命令类型	说 明
0000	中断响应	中断识别命令
0001	特殊周期	提供在总线上的简单广播机制
0010	I/O 读	
0011	I/O 写	
0100	保留	
0101	保留	
0110	存储器读	
0111	存储器写	
1000	保留	
1001	保留	
1010	读配置	用来读每一个主控器的配置空间
1011	写配置	用来写每一个主控器的配置空间
1100	存储器重复读	只要 FRAME 有效,就应保持流水线的连续,以便传送大量的数据
1101	双地址节拍	用来传送 64 位地址到某一设备
1110	高速缓存读	用于多余两个 32 位的数据期
1111	高速缓存写	

PAR(Parity)：对 $AD_{31} \sim AD_0$ 和 $C/\overline{BE}_3 \sim C/\overline{BE}_0$ 信号作奇偶校验(偶校验),以保证数据的有效性。

(2) 接口控制信号

\overline{FRAME} ：帧周期信号。由当前总线主设备驱动,表示一个总线周期的开始和结束。

\overline{TRDY} ：从设备准备好信号。由从设备驱动,表示从设备准备好传送数据。

\overline{IRDY} ：主设备准备好信号。由系统主设备驱动,与 \overline{TRDY} 信号同时有效可完成数据传输。

\overline{STOP} ：停止信号。从设备要求主设备停止当前数据传送。

\overline{DEVSEL} ：设备选择信号。该信号有效时(输出),表示所译码的地址是在设备的地址范围内,当作输入信号时,表示总线上有某设备是否被选中。

\overline{IDSEL} ：初始化设备选择信号。在配置读写期间,用作芯片选择。

\overline{LOCK} ：锁定信号。用于保证主设备对存储器的锁定操作。

(3) 错误报告信号

$\overline{\text{PERR}}$ (Parity Error): 数据奇偶校验错信号。

$\overline{\text{SERR}}$ (System Error): 系统错误信息。用于报告地址奇偶错、数据奇偶错和命令错等。

(4) 总线仲裁信号

$\overline{\text{REQ}}$ (Request): 总线请求信号。由希望成为总线主设备的设备驱动,是一个点对点的信号。

$\overline{\text{GNT}}$ (Grant): 总线请求允许信号。

(5) 系统信号

CLK : 总线时钟信号。它是系统时钟信号,该信号频率为 PCI 总线的工作频率。

$\overline{\text{RST}}$: 系统复位信号。它有效时,PCI 总线的所有输出信号处于高阻状态。

(6) 64 位扩展信号

$\text{AD}_{63} \sim \text{AD}_{32}$: 地址数据扩展信号。

$\text{C}/\overline{\text{BE}}_7 \sim \text{C}/\overline{\text{BE}}_1$: 高 32 位地址命令/字节允许信号。

PAR_{64} : 高 32 位奇偶校验信号。

REQ_{64} : 64 位传送请求信号。

ACK_{64} : 64 位传送响应信号。

(7) 中断请求信号

$\overline{\text{INTX}}$: 中断请求信号, $X = A, B, C, D$ 。

(8) Cache 支持信号

$\overline{\text{SBO}}$ (Snoop Backoff): 探测返回信号。有效时,关闭预测命令中的一个缓冲行。

$\overline{\text{SDONE}}$ (Snoop Done): 探测完成信号。有效时,表示探测完成,命中一个缓冲行。

(9) JTAG 边界扫描测试引脚

JTAG 提供了板级和芯片级的测试,通过定义输入输出引脚,逻辑扩展函数和指令,所有 JTAG 的测试功能仅需一个 4 线或 5 线的接口,以及相应软件即可完成。利用 JTAG 可测试电路板的连接和功能。

JTAG 是 PCI 总线的一个可选接口。有 5 个信号:

TCK (Test Clock): 测试时钟。用于控制状态机及数据传送。

TDI (Test Data In): 测试数据输入。用于 TCK 上升沿接受 JTAG 串行指令和数据。

TDO (Test Data Out): 测试数据输出。用于 TCK 下降沿输出 JTAG 串行数据。

TMS (Test Mode Select): 测试模式选择。用于控制边界扫描模式,控制状态机的测试操作。

$\overline{\text{TRST}}$ (Test Reset): 测试复位。

3. PCI 总线引脚定义

PCI 总线有 124 个信号线用于连接 PCI 卡,PCI 卡的总线连接头上每面各有 62 个引线,PCI 总线引脚信号定义如表 6-2 所示。

表 6-2 PCI 总线引脚定义

引 脚	B 面	A 面	引 脚	B 面	A 面
1	-12V	$\overline{\text{TRST}}$	32	AD ₁₇	AD ₁₆
2	TCK	+12V	33	C/ $\overline{\text{BE}}_2$	+3.3V
3	GND	TMS	34	GND	$\overline{\text{FRAME}}$
4	TDO	TDI	35	$\overline{\text{IRDY}}$	GND
5	+5V	+5V	36	+3.3V	$\overline{\text{TRDY}}$
6	+5V	$\overline{\text{INTA}}$	37	$\overline{\text{DEVSEL}}$	GND
7	$\overline{\text{INTB}}$	$\overline{\text{INTC}}$	38	GND	$\overline{\text{STOP}}$
8	$\overline{\text{INTD}}$	+5V	39	$\overline{\text{LOCK}}$	+3.3V
9	PRSNT ₁	Reserved	40	PREE	SDONE
10	Reserved	+5V	41	+3.3V	$\overline{\text{SBO}}$
11	$\overline{\text{PRSNT}}_2$	Reserved	42	$\overline{\text{SERR}}$	GND
12	GND	GND	43	+3.3V	PAR
13	GND	GND	44	C/ $\overline{\text{BE}}_1$	AD ₁₅
14	Reserved	Reserved	45	AD ₁₄	+3.3V
15	GND	RST	46	GND	AD ₁₃
16	CLK	+5V	47	AD ₁₂	AD ₁₁
17	GND	GNT	48	AD ₁₀	GND
18	REQ	GND	49	GND	AD ₀₉
19	+5V	Reserved	50	Keyway	连接器空位槽
20	AD ₃₁	AD ₃₀	51	Keyway	连接器空位槽
21	AD ₂₉	+3.3V	52	AD ₀₈	C/ $\overline{\text{BE}}_0$
22	GND	AD ₂₈	53	AD ₀₇	+3.3V
23	AD ₂₇	AD ₂₆	54	+3.3V	AD ₀₆
24	AD ₂₅	GND	55	AD ₀₅	AD ₀₄
25	+3.3V	AD ₂₄	56	AD ₀₃	GND
26	C/ $\overline{\text{BE}}_3$	$\overline{\text{IDSEL}}$	57	GND	AD ₀₂
27	AD ₂₃	+3.3V	58	AD ₀₁	AD ₀₀
28	GND	AD ₂₂	59	+5V	+5V
29	AD ₂₁	AD ₂₀	60	$\overline{\text{ACK}}_{64}$	$\overline{\text{REG}}_{64}$
30	AD ₁₉	GND	61	+5V	+5V
31	+3.3V	AD ₁₈	62	+5V	+5V

6.3 通用外部总线标准

计算机与计算机之间、计算机和一部分外设之间常用的通信方式有两种，即并行通信方式和串行通信方式。对应这两种通信方式，通信总线也有两种，即并行通信总线和串行通信总线。它们不仅用于微型计算机系统中，还广泛用于技术网络、远程检测系统、远程控制系统和各种电子设备中，被统称为通用的外部总线。

对于微型计算机系统来说，外部通用的总线标准除了最简单的 RS-232C 和打印机专

用的 Centronics 总线标准外,最常用的就是 IDE(Intelligent Device Electronic)、SCSI(Small Computer System Interface)和 USB(Universal Serial Bus)总线标准。

6.3.1 并行 I/O 标准接口 IDE 和 EIDE

IDE 总线是 Compaq 公司联合 Western Digital 公司专门为主机和硬盘子系统连接而设计的外部总线,也适用于和软盘、光驱的连接,IDE 也称为 ATA(AT Attachable)接口。目前,在微型计算机系统中,主机和硬盘子系统之间都采用 IDE 或 EIDE 总线连接。

采用 IDE 接口以后,硬盘控制器和驱动器组合在一起,主机和硬盘子系统用 40 芯的扁平电缆连接在一起。这样,不仅省下了一个插槽,而且使驱动器和控制器之间传输距离大大缩短,从而提高了可靠性,有利于数据传输速度的提高。

IDE 采用 16 位并行传输,其中除数据线外,还有一组 DMA 请求和应答信号、1 个中断请求信号、I/O 读信号、I/O 写信号、复位信号和地信号等。同时,IDE 另用 1 个 4 芯电缆将主机的电源送往外设子系统。通常情况下,IDE 的传输率为 8.33MB/s,每个硬盘的最高容量为 528MB。

6.3.2 并行 I/O 标准接口 SCSI

SCSI 是小型计算机系统接口的简称,其设计思想来源于 IBM 大型机系统的 I/O 通道结构,目的是使 CPU 摆脱对各种设备的繁杂控制。它是一个高速智能接口,可以混接各种磁盘、光盘、磁带机、打印机、扫描仪、条码阅读器以及通信设备。它首先应用于 Macintosh 和 Sun 平台上,后来发展到工作站、网络服务器和 Pentium 系统中,并成为 ANSI(美国国家标准局)标准。在微型计算机系统中用得较少。

SCSI 有如下性能特点:

① SCSI 接口总线由 8 条数据线、一条奇偶校验线、9 条控制线组成。使用 50 芯电缆,规定了两种电气条件:单端驱动,电缆长 6m;差分驱动,电缆最长 25m。

② 总线时钟频率为 5MHz,异步方式数据传输率是 2.5MB/s,同步方式数据传输率是 5MB/s。

③ SCSI 接口总线以菊花链形式最多可连接 8 台设备。在 Pentium 中通常是:由一个主适配器 HBA 与最多 7 台外围设备相连,HBA 也算作一个 SCSI 设备,由 HBA 经系统总线(如 PCI)与 CPU 相连。

④ 每个 SCSI 设备有自己唯一的设备号。ID=7 的设备具有最高优先权,ID=0 的设备优先权最低。SCSI 采用分布式总线仲裁策略。在仲裁阶段,竞争的设备以自己的设备号驱动数据线中相应的位线(如 ID=7 的设备驱动 DB_7 线),并与数据线上的值进行比较。因此仲裁逻辑比较简单,而且在 SCSI 的总线选择阶段,启动设备和目标设备的设备号能同时出现在数据线上。

⑤ 所谓 SCSI 设备是指连接在 SCSI 总线上的智能设备,即除了主适配器 HBA 外,其他 SCSI 设备实际是外围设备的适配器或控制器。每个适配器或控制器通过各自的设备级 I/O 线可连接一台或几台同类型的外围设备(如一个 SCSI 磁盘控制器接两台硬盘

驱动器)。标准允许每个 SCSI 设备最多有 8 个逻辑单元,每个逻辑单元可以是物理设备也可以是虚拟设备。每个逻辑单元有一个逻辑单元号(LUN₀~LUN₇)。

⑥ 由于 SCSI 设备是智能设备,对 SCSI 总线乃至主机屏蔽了实际外设的固有物理属性(如磁盘柱面数、磁头数等参数),各 SCSI 设备之间就可用一套标准的命令进行数据传送,也为设备的升级或系统的系列化提供了灵活的处理手段。

⑦ SCSI 设备之间是一种对等关系,而不是主从关系。SCSI 设备分为启动设备(发命令的设备)和目标设备(接受并响应命令的设备)。但启动设备和目标设备是依当时总线运行状态来划分的,而不是预先规定的。

总之,SCSI 是系统级接口,是处于主适配器和智能设备控制器之间的并行 I/O 接口。一块主适配器可以接 7 台具有 SCSI 接口的设备,这些设备的类型可以完全不同,主适配器只占主机的一个槽口。这可以缓解计算机挂接外设的数量和类型越来越多、主机槽口日益紧张的状况。

为提高数据传输率,改善接口的兼容性,20 世纪 90 年代又陆续推出了 SCSI-2 和 SCSI-3 标准。SCSI-2 扩充了 SCSI 的命令集,提高了时钟速率和数据线宽度,最高数据传输率可达 40MB/s,采用 68 芯电缆,且对电缆采用有源终端器。SCSI-3 标准允许 SCSI 总线上连接的设备由 8 个提高到 16 个,可支持 16 位数据传输。另一个变化就是发展串行 SCSI,使串行数据传输率达到 640MB/s(电缆)或 1GB/s(光纤),从而使串行 SCSI 成为 IEEE1394 标准的基础。

6.3.3 通用串行总线 USB

USB 是 Intel、DEC、Compaq、Microsoft 和 IBM 等公司 1996 年共同制定的串行接口标准,其设计初衷是作为一种通用的串行总线,能够用一个 USB 端口连接所有不带适配卡的外设,提供所谓“万用”(one size fits all)连接功能。而且可以在不开机箱的情况下增减设备,支持即插即用功能。

USB 的连接方式很简单,只用一条四芯电缆,除了电源和地以外,用两根信号线以差分方式串行传输数据,连线可长达 5m。USB 可用菊花链式或集线器式两种方式连接多台设备,前者是链式扩展的,可连接多台外设,而后者是星型扩展的,可连接多达 127 台外设。

USB 适用于不同的设备要求。可连接键盘、鼠标、移动盘、Modem、扫描仪、数码相机和打印机等外设。通常,键盘和鼠标用低速率,移动盘、扫描仪、数码相机和打印机等用中速率。USB 可使中速、低速的串行外设很方便地与主机相连接,不需要另加接口卡,并在软件配合下支持即插即用功能。不过,USB 对硬件和软件两方面都提出了要求,硬件上,CPU 必须为 Pentium 以上的芯片,软件上,必须为 Windows 98 以上的版本。1996 年推出的是 USB 1.0 版本规范,2000 年 4 月又推出了 USB 2.0 版,既支持更高性能的外设的连接,也支持低速外设的连接。现在的 PC 机大多配备了 USB 功能,而且市场上采用 USB 接口的外设越来越多,价格也较低廉。随着 USB 2.0 输入输出带宽的显著提高,进一步刺激了 USB 外设的发展,随着新标准的推出,用户很快就可享受更快的宽带 Internet 接口、分辨率更高的电视会议摄影机、新一代打印机和扫描仪以及更快的外置存

储设备。

1. USB 的特点

USB 之所以能被大家广泛接受,主要是其有以下主要特点:

① 速度快。USB1.1 接口支持的数据传输率最高为 12Mb/s,USB 2.0 接口支持的传输速度高达 480Mb/s。

② 连接简单快捷,可进行热插拔。USB 接口设备的安装非常简单,在计算机正常工作时也可以进行安装,无须关机、重新启动或打开机箱等操作。

③ 无须外接电源。USB 提供内置电源,能向低压设备提供 +5V 的电源,使得系统不用另外配备专门的交流电源以供新增外设使用。

④ 扩充能力强。USB 支持多设备连接,减少了 PC 机 I/O 口的数量,避免了 PC 机插槽数量对扩充外设的限制以及如何配置系统资源的问题。使用设备插架技术最多可扩充 127 个外围设备。

⑤ 具有高保真音频。在使用 USB 音箱时,由于是在计算机外生成 USB 的音频信息,从而减少了电子噪声对声音质量的干扰,使系统具有较高的保真度。

⑥ 良好的兼容性。USB 接口标准具有良好的向下兼容性,以 USB 2.0 和 USB 1.1 标准为例,USB 2.0 标准就能很好地兼容以前的 USB 1.1 的产品。系统在自动检测到 1.1 版本的接口类型时,会自动按照以前的 12Mb/s 的速度进行传输,而其他的采用 USB 2.0 标准的设备,并不会因为接入了一个 USB 1.1 标准的设备,而减慢它们的速度,它们还是能以 USB 2.0 标准所规定的速度进行传输。

2. USB 的连接方法

USB 提供中、低速率外设装置的扩充能力,这些中、低速的外设都可通过 USB 与 PC 机(及其他计算机系统)连接并传送数据,不需要搭配附加的接口卡来占用 PC 机的扩展槽。USB 设备的连接如图 6-3 所示。

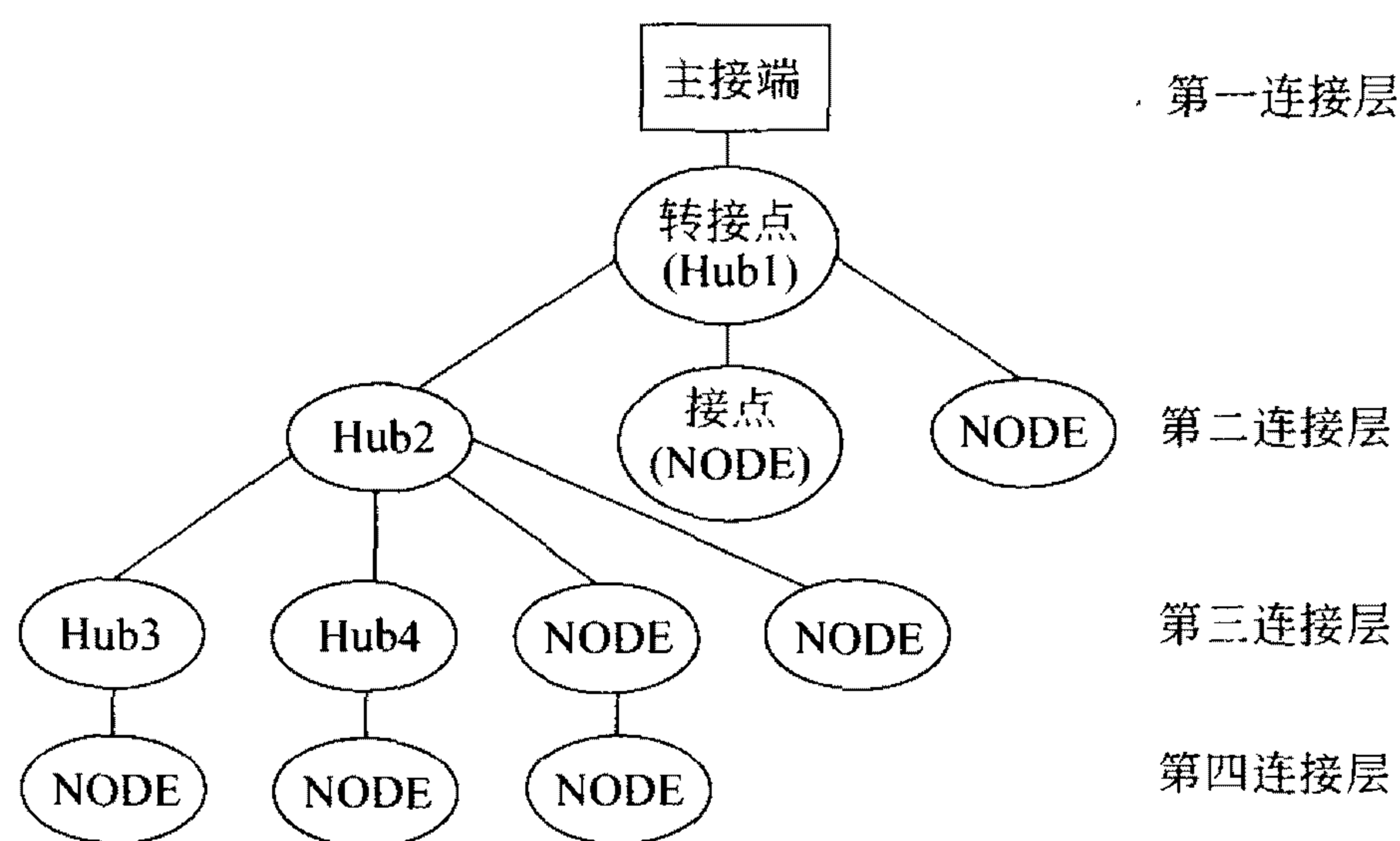


图 6-3 USB 设备连接图

USB 设备以转接器(Hub)与设备节点(NODE)的方式连接的,最多可以延伸到 4 个

层次。PC 机主板上一般最少配备 2 个 USB 连接器,可以连接 2 个或多个 USB 设备,其中一个可接到 Hub 或具有 Hub 功能的 USB 设备。每一个 Hub 至少提供 2 个连接器以及连接到下一个 Hub 的能力。因此,用户要安装 Hub 设备时,只要找到一个 Hub(转接器)底下的连接孔,把 Hub 设备的插头直接插入即可。这样,一个 USB 系统从整体上可看成一种树形结构。按 USB 规格设计,最多可以同时使用 127 个外设(含 Hub)在同一台 PC 机上。

3. USB 的接口设计

(1) USB 系统组成

完整的 USB 体系如图 6-4 所示。

最底层的是 USB 设备,往上是 USB 主机控制器,这些是 USB 的硬件部分。

然后就是软件部分,首先是 USB 主机控制器驱动程序(Host Contrllet Driver)。Windows 95 OSR2.1 以上版本及 Windows 98 提供了这个最低层的驱动程序。其他的不一定需要操作系统支持,只要主板芯片组的开发商提供了南桥的 USB 驱动程序也可以使用 USB 设备。但在安装芯片组的驱动程序时,如果安装程序发现操作系统是 Windows 95 OSR2.1 以上版本,就会警告用户无须安装驱动程序,因为操作系统已经自带了。

再往上是 USB 设备驱动(USB Device Driver)程序。众所周知,没有驱动程序,硬件是不能使用的。Windows 95/98 已经内建了一些常用的 USB 设备的驱动程序,如 USB 音箱、USB Hub 等。其他的 USB 设备驱动程序,操作系统并没有内建,由生产商附送,如 USB 摄像头、USB MODEM 等。Windows ME 和 Windows XP 内建了更多的 USB 设备的驱动程序,使 USB 设备使用更方便。

最后就是 USB 应用程序(USB Application,或者叫 Client Driver software)。USB 设备需要有相应的应用程序才能发挥作用。像 USB 扫描仪就必须有扫描程序才能使用,而 USB 接口的数码相机也需要相应的应用程序才能传输相片等资料。

(2) USB 的硬件结构

USB 采用 4 线电缆,其中 2 根是用来传送数据的串行通道,另 2 根为下游(Downstream)设备提供电源,如图 6-5 所示。

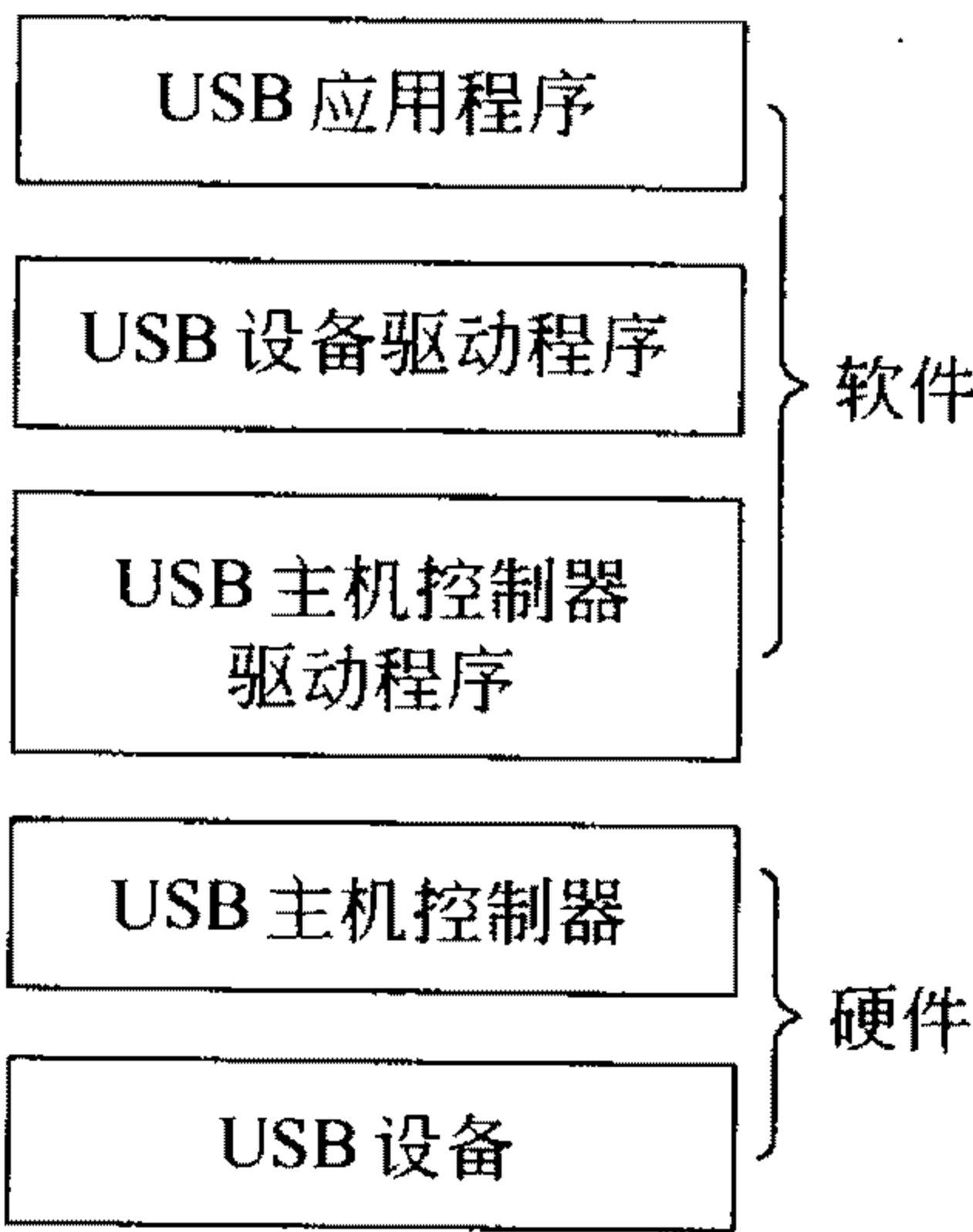


图 6-4 USB 体系

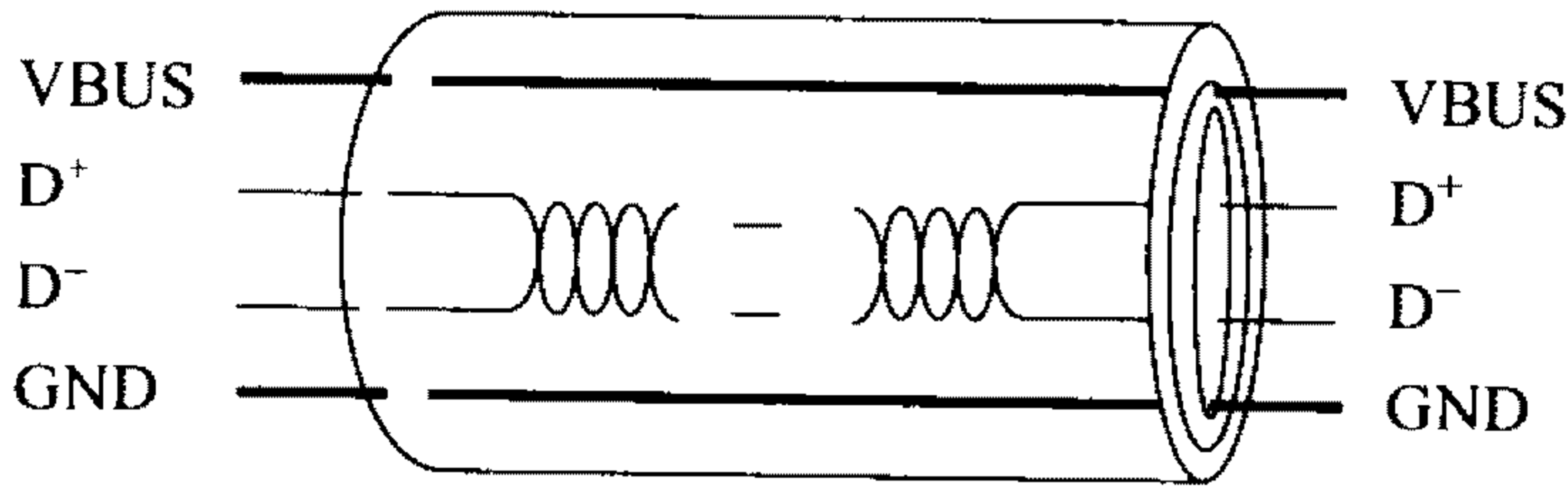


图 6-5 USB 硬件结构

其中, D^+ 和 D^- 是串行数据通信线, 对 USB 1.1 版本而言, 它支持两种数据传输率, 对于高速且需要高带宽的外设, USB 以全速 12Mb/s 传输数据; 对于低速外设, USB 则以 1.5Mb/s 的传输速率传输数据。USB 总线会根据外设情况在两种传输模式中自动进行动态转换。VBUS 为 +5V 电源, GND 是地线。USB 是基于令牌的总线, 类似于令牌环网络或 FDDI 基于令牌的总线。USB 主控制器广播令牌, 总线上设备检测令牌中的地址是否与自身相符, 通过接收或发送数据给主机来响应, USB 通过支持悬挂/恢复操作来管理 USB 总线电源。而 USB 2.0 版本的使用与 USB 1.1 版本的使用相仿。

USB 系统采用级联星型拓扑结构连接, 即类似菊花链连接。该拓扑结构由 3 个基本部分组成: 主机(Host)、集线器(Hub)和功能设备(USB 设备)。

主机, 也被称为根、根结或根 Hub。主机包含有主控制器和根集线器(Root Hub), 控制着 USB 总线上的数据和控制信息的流动。每个 USB 系统只能有一个根集线器, 它连接在主控制器上。

集线器是 USB 结构中的特定成分, 它提供叫做端口(Port)的点将设备连接到 USB 总线上, 同时检测连接在总线上的设备, 并为这些设备提供电源管理, 负责总线的故障检测和恢复。集线器或是为总线提供能源, 或是为自身提供能源(从外部得到电源)。自身提供能源的设备可插入总线提供能源的集线器中, 但总线提供能源的设备不能插入自身提供能源的集线器。总线提供能源的设备需要超过 100mA 的电流时, 不能同总线提供电源的集线器连接。

功能设备通过端口与总线连接。USB 设备同时可做 Hub 使用。例如, USB 监视器可以提供 USB 鼠标和 USB 键盘的端口。USB 集线器使用 A 类连接器(扁平形), USB 设备使用 B 类连接器(方形)。

4. USB 的软件结构

USB 通信模块的基本流程如图 6-6 所示。

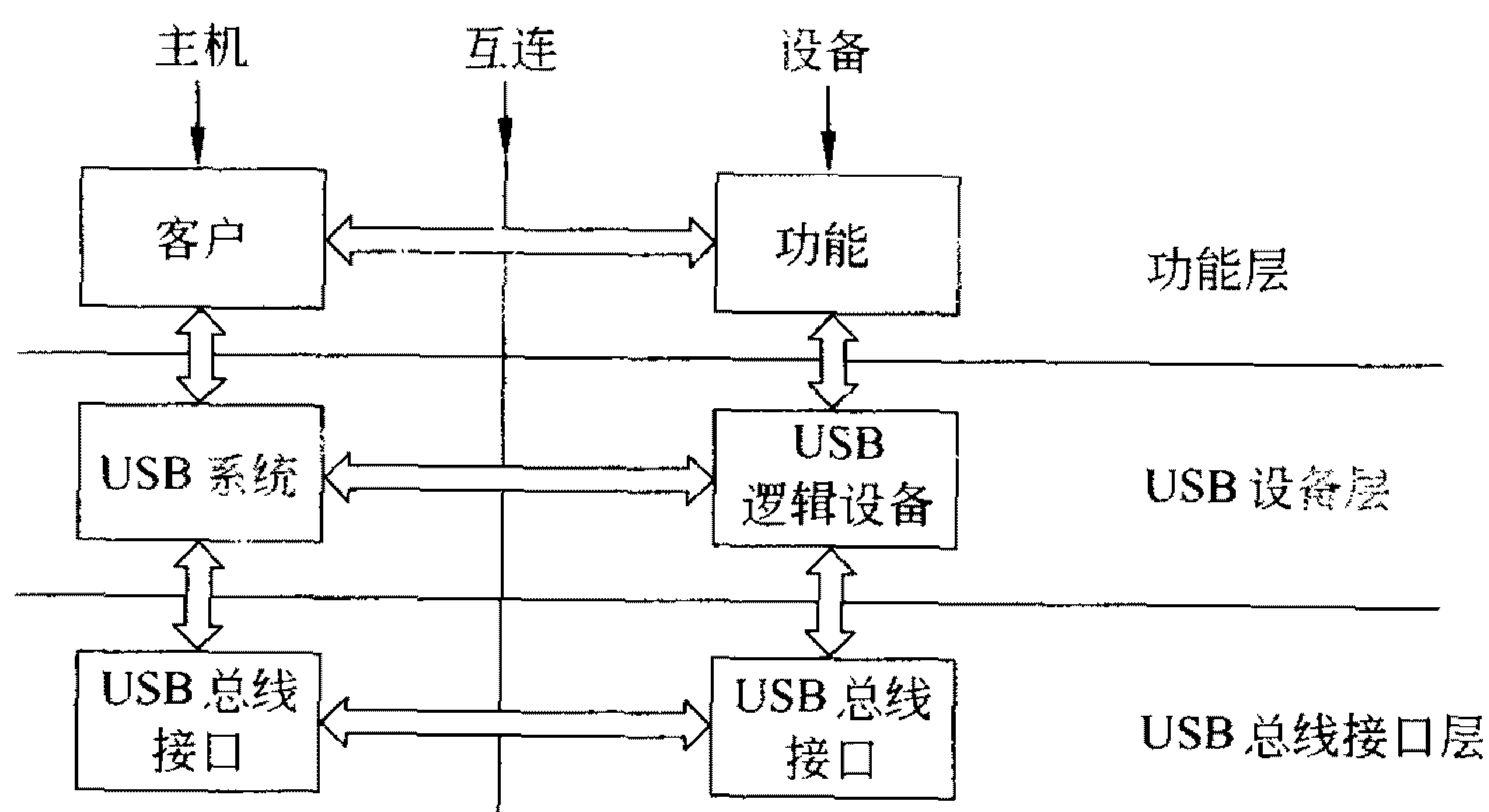


图 6-6 USB 通信模块基本流程

主机和设备被分为如图 6-6 所示的几层。黑箭头表示主机上的实际通信。设备上的相应接口根据不同的仪器而不同。主机和设备间的通信最终发生在物理线上, 但在每一水平层之间存在着逻辑接口。主机中客户程序软件与设备功能间的通信代表了设备需

求与设备能力之间的约定。

每个 USB 只有一个主机,它包括以下几层:

(1) USB 总线接口

USB 总线接口处理电气层与协议层的互连。从互连的角度来看,相似的总线接口由设备及主机同时给出,例如串行接口机(SIE)。USB 总线接口由主控制器实现。

(2) USB 系统

USB 系统用主控制器管理主机与 USB 设备间的数据传输。它与主控制器间的接口依赖于主控制器的硬件定义。同时,USB 系统也负责管理 USB 资源,例如带宽和总线能量,这使客户访问 USB 成为可能。USB 系统有 3 个基本组件:

- 主控制器驱动程序(HCD)

HCD 能够更容易地将不同主控制器设备映射到 USB 系统中。因此,客户可以在不知其设备连接哪个主控制器的情况下,与设备相互作用。HCD 与 USB 的接口叫 HC DI,特定的 HC DI 由支持不同主控制器的操作系统定义。通用主控制器驱动器(UHCD)处于软结构的最底层,由它来管理和控制主控制器。USB 主控制器定义了一个标准硬件接口,以提供一个统一的主控制器可编程接口。UHCD 实现与 USB 主控制器通信和控制 USB 主控制器的一些细节,并且它对系统软件的其他部分是隐蔽的。系统软件中的更高层通过 UHCD 的软件接口与主控制器通信。

- USB 驱动程序(USBD)

位于 UHCD 之上,它提供驱动器级的接口,满足现有设备驱动器设计的要求。USBD 所实现的准确细节随操作系统环境的不同而有所不同,但 USBD 在不同操作系统环境下完成的是一样的工作。USBD 以 I/O 请求包(IRPs)的形式提供数据传输架构,它由通过特定管道(Pipe)传输数据的需求组成。此外,USBD 使客户端出现设备的一个抽象,以便于抽象和管理。作为抽象的一部分,USBD 拥有默认的管道。通过它可以访问所有的 USB 设备以进行标准的 USB 控制。该默认管道描述了一条 USBD 和 USB 设备间通信的逻辑通道。

- 主机软件

在某些操作系统中,没有提供 USB 系统软件。这些软件本来是用于向设备驱动程序提供配置信息和装载结构的。在这些操作系统中,设备驱动程序将应用提供给接口而不是直接访问 USBDI(USB 驱动程序接口)结构。

(3) USB 客户软件

它位于软件结构的最高层,负责处理特定 USB 设备的设备驱动器。客户程序层描述了所有直接作用于设备的软件入口。当设备被系统检测到后,这些客户程序将直接作用于外围硬件。这个共享的特性将 USB 系统软件置于客户和它的设备之间,也就是说,一个客户程序不能直接访问硬件设备,而要根据 USBD 在客户端形成的设备映像由客户程序对它进行处理。

总体上说,主机各层有以下功能:

- ① 检测连接和移去的 USB 设备;
- ② 管理主机和 USB 设备间的数据流;

③ 连接 USB 的状态和活动的统计;

④ 控制主控制器和 USB 设备间的电气接口,包括限量能量供应。

控制信息可能以带内方式或带外方式在主机和设备间传输。带内方式将控制信息与数据混在一个管道内;带外方式将控制信息与数据放在分离管道内。

每个连接上的 USB 设备都有一个被称为默认管道的消息管道,并在 USB 设备和主机之间建立逻辑关联。默认管道为所有的设备提供了一个标准的接口。默认通道也用于设备通信,由 USB-D 作为中介,USB-D 拥有所有设备的默认通道。

特别的 USB 设备允许使用附加的消息管道传输具体设备的控制信息。这些管道使用相同的通信协议作为默认通道,但传输的信息必须具体到特定的设备,而不被规范标准化。USB-D 支持其客户共享它拥有和使用的默认通道,它也可以访问其他设备的控制管道。

基于不同级别的抽象,HCDI 和 USB-DI 提供不同的软件接口。它们以某种特殊的方式一起工作来满足所有 USB 系统的需求。USB 系统的需求主要体现为对 USB-DI 的需求。USB-D 和 HCD 间任务的区分没有定义,然而,在特定的操作系统中支持多主控制器设备是 HCDI 必须满足的需求。

HCD 提供了主控制器的抽象和通过 USB 传输的数据的主控制器视角的一个抽象。USB-D 提供了 USB 设备的抽象和 USB-D 客户与 USB 功能间数据传输的一个抽象。总之,USB 系统促进了客户和功能间的数据传输,并作为 USB 设备的规范接口的一个控制点。USB 系统提供缓冲区管理能力并允许数据传输同步于客户和功能的需求。

5. USB 上的数据流传输

主控制器负责主机和 USB 设备间数据流的传输。这些传输数据被当作连续的比特流。每个设备提供了一个或多个可以与客户程序通信的接口,每个接口由 0 个或多个管道组成,这些管道分别独立地在客户程序和设备的特定终端间传输数据。USB-D 为主机软件的实际需求建立了接口和管道,当提出配置请求时,主控制器根据主机软件提供的参数提供服务。

USB 支持 4 种基本的数据传输模式:控制传输、等时传输、中断传输和数据块传输。每种传输模式应用到具有相同名字的终端,则具有不同的性质。

控制传输类型支持外设与主机之间的控制、状态和配置等信息的传输,为外设与主机之间提供一个控制通道。每种外设都支持控制传输类型,这样,主机与外设之间就可以传送配置和命令/状态信息。

等时(Isochronous)传输类型支持具有周期性、有限的时延和带宽且数据传输速率不变的外设与主机间的数据传输。该类型无差错校验,故不能保证正确的数据传输,支持计算机-电话集成系统(CTI)和音频系统与主机的数据传输。

中断传输类型支持像游戏棒、鼠标和键盘等人机输入设备,这些设备与主机间数据传输量小、无周期性,但对响应时间敏感,要求立即响应。

数据块(Bulk)传输类型支持打印机、扫描仪和数码相机等外设,这些外设与主机间传输的数据量很大,USB 在满足带宽的情况下才进行该类型的数据传输。

USB 采用分块带宽分配方案,若外设超过当前带宽分配或潜在的要求,则拒绝进入该设备。同步和中断传输类型的终端,保留带宽并保证数据按一定的速率传送。集中和控制终端按可用的最佳带宽来传输数据。但是,10%的带宽为批量处理和控制传送而保留,数据块传输仅在带宽满足要求的情况下才会出现。

6. USB 的即插即用

USB 的一个主要优点就是支持设备的热插拔,用户不需要关闭电源就可以接上和使用 USB 设备。USB 集线器驱动程序检测设备,并通知系统,设备就绪。USB 设备使用描述符来识别设备及其使用协议。串口号产生 P&P 用的 ID,端口地址指明设备连接的端口和集线器。若设备不提供串口号,则 USB 使用设备端口地址。

当一个新设备被 USB 集线器检测到后,马上通知主系统,系统软件即查询该设备,自动确定所需设备驱动软件和总线带宽,然后对其进行配置。系统软件装载了合适的驱动程序后,用户马上就可以使用新设备。

需要说明的是,随着要求高速数据传输速率的外围设备(如 CD 播放机、电视机顶盒和数字化摄像录像一体机等)的增多,USB 将无法满足其高速的要求。于是出现了 IEEE 1394(又称 Fire Wire,火线),它是由 Apple 公司和 TI 公司开发的高速串行接口标准,最高数据传输率可达 1Gb/s(1024Mb/s)。它具有把一个输入信息源传来的数据向多个输出机器广播的功能,特别适用于家庭视听 AV(Audio-Visual)的连接。在多媒体信息处理系统中,IEEE 1394 是更有前途的串行接口标准。

6.3.4 视频接口 AGP

加速图形端口 AGP(Accelerated Graphics Port)是在三维图形显示中,为解决“图形纹理”数据高速传输的瓶颈问题而产生的。

在 20 世纪 90 年代后期,计算机中的主存与图形卡之间是用 PCI 总线连接的,其最大的数据传输率为 133MB/s。同时,由于硬盘控制器、LAN(局域网)卡和声卡等都是通过 PCI 总线同主存交换数据的。因此,实际的数据传输率远低于 133MB/s。而 AGP 卡在三维图形处理时不仅要求惊人的数据量,而且要求更宽广的数据传输频宽。例如,对 640×480 的分辨率而言,以每秒 75 次画面更新率来计算,“画面输出”需 50MB/s、色彩输出需 100MB/s、Z 轴缓冲区需 100MB/s、“贴图纹理”需 100MB/s,其他开销需 20MB/s,则要求全部的数据频宽高达 370MB/s;若分辨率提高到 800×600 ,总频宽流量为 580MB/s;若显示器分辨率提高到 1024×768 ,则总频宽要求更高。原有计算机中 133MB/s 数据传输率的 PCI 总线就成为三维图形加速卡上高速传送图形纹理数据的一大瓶颈。

AGP 是 Intel 公司开发的于 1996 年 7 月正式公布的一种新型视频接口技术标准,它定义了一种超高速的连通结构,把三维图形控制器从 PCI 总线上卸下来,用专用的点对点通道——AGP,把图形控制器直接连在控制芯片组(“主存/PCI”控制芯片组)上,三维图形芯片可以将主存作为帧缓冲器,实现高速存取。AGP 直接连通的系统芯片组以 66.7MHz 直接同主存联系,而 AGP 的数据宽度为 32 位,因此它的最大数据传输量为

$4 \times 66.7 = 226\text{MB/s}$, 是传统的 PCI 总线频宽的 2 倍。另外 AGP 还定义了一种“双激励”的传输技术, 它能够在时钟的上、下边沿由双向传递数据, 这样 AGP 实现数据传输的频率就变成 $2 \times 66.7\text{MHz}$ 即 133MHz , 而其最大数据传输量也增为 $4 \times 133\text{MB/s} = 533\text{MB/s}$ 。上述第一种情况 (66.7MHz 时钟) 称为“基线 AGP”或“AGP-1 \times ”, 第二种情况 (双激励) 称为“全 AGP”或“AGP-2 \times ”。当采用 AGP 2.0 技术后, AGP 的时钟频率为 133MHz , 有效频宽将为 1GB/s , 是传统 PCI 的 8 倍, 成为“AGP-4 \times ”。

AGP 的地址和数据线分离, 没有“切换”的“开销”, 提高了随机访问主存时的性能。AGP 可实现“流水线”处理, 提高了数据传输速率。同时 AGP 是图形加速卡的一条专用信息通道, 不用与其他任何设备共享, 任何时候想调用该信息通道都会立即得到响应, 效率极高。另外, 由于将图形加速卡从 PCI 上分离出来, 使 PCI 总线上的其他设备的工作效率随之提高, PCI 总线的重负载得到缓解。

AGP 接口除可以采用直接存储器存取 (Direct Memory Access, DMA) 传输图形数据外, 还支持直接内存执行 (Direct Memory Execute, DME) 方式。后者可以直接在系统内存中处理图形数据, 而不需要将原始数据全部传输到图形加速显示卡中, 这样极大地减少了数据传输量, 提高了性能。

以上都是 AGP 在技术上的优点所在, 而从开发方面考虑, AGP 能配置在低价位的计算机中, 它使相应的器件 (图形控制芯片) 制造简单, 成本较低。由于 AGP 除同主存/PCI 控制芯片组连接, 只限于连接一个器件, 因此所连接的器件容易开发, 在主存/PCI 控制芯片组, 不必安装用于 AGP 仲裁的专用电路, 同样可以降低开发成本。

6.4 32 位微型计算机总线结构

PC 系列机采用开放式的总线结构。随着计算机技术的发展和应用的推广, 涌现了一大批总线标准, 使计算机的总线结构逐步规范化、通用化, 用户可按功能需要选用不同厂家生产的、基于同种总线标准的模块和设备, 组建符合自己要求的应用系统, 保证了各级产品 (芯片、模块、设备等) 的兼容性、互换性, 以及整个系统的可维护性和可扩充性。

现代 32 位微型计算机系统的体系结构根据微处理器的不同而有所区别。80386/80486 微处理器的微型计算机体系结构, 主要采用 ISA 总线和 EISA 总线结构, 后期的 80486 还采用了 VESA 总线、EISA 总线和 PCI 总线作为各个部件的连线。采用 Pentium 微处理器的微型计算机总线结构, 最主要的表现就是主板的总线结构各不相同。为了提高微型计算机的整体性能, 规范体系结构的接口标准, 根据各部件处理或传输信息的速度快慢, 采用了更加明显的三级总线结构, 即 CPU 总线 (Host Bus)、局部总线 (PCI 总线) 和系统结构总线 (一般是 ISA 总线)。其中, CPU 总线是 64 位数据线、32 位地址线的同步总线; PCI 总线为 32 位或 64 位数据/地址分时复用同步总线, 它作为高速的外围总线不仅能够直接连接高速的外围设备, 而且通过桥芯片和更高速的 CPU 总线与系统总线相连。系统总线仍为 16 位数据线、20 位地址线。三级总线之间由集成度更高的多功能桥路芯片组成的芯片组相连, 形成一个统一的整体。这些桥路芯片起到信号缓冲、电平转换和控制协议转换的作用。

图 6-7 所示的就是采用 PCI 局部总线的 Pentium 微型计算机的体系结构。在这种结构中,主要通过两个桥片将三级总线连接起来。这两个桥片就是被称为北桥的 CPU 总线-PCI 桥片(Host Bridge)和被称为南桥的 PCI-ISA 桥片。

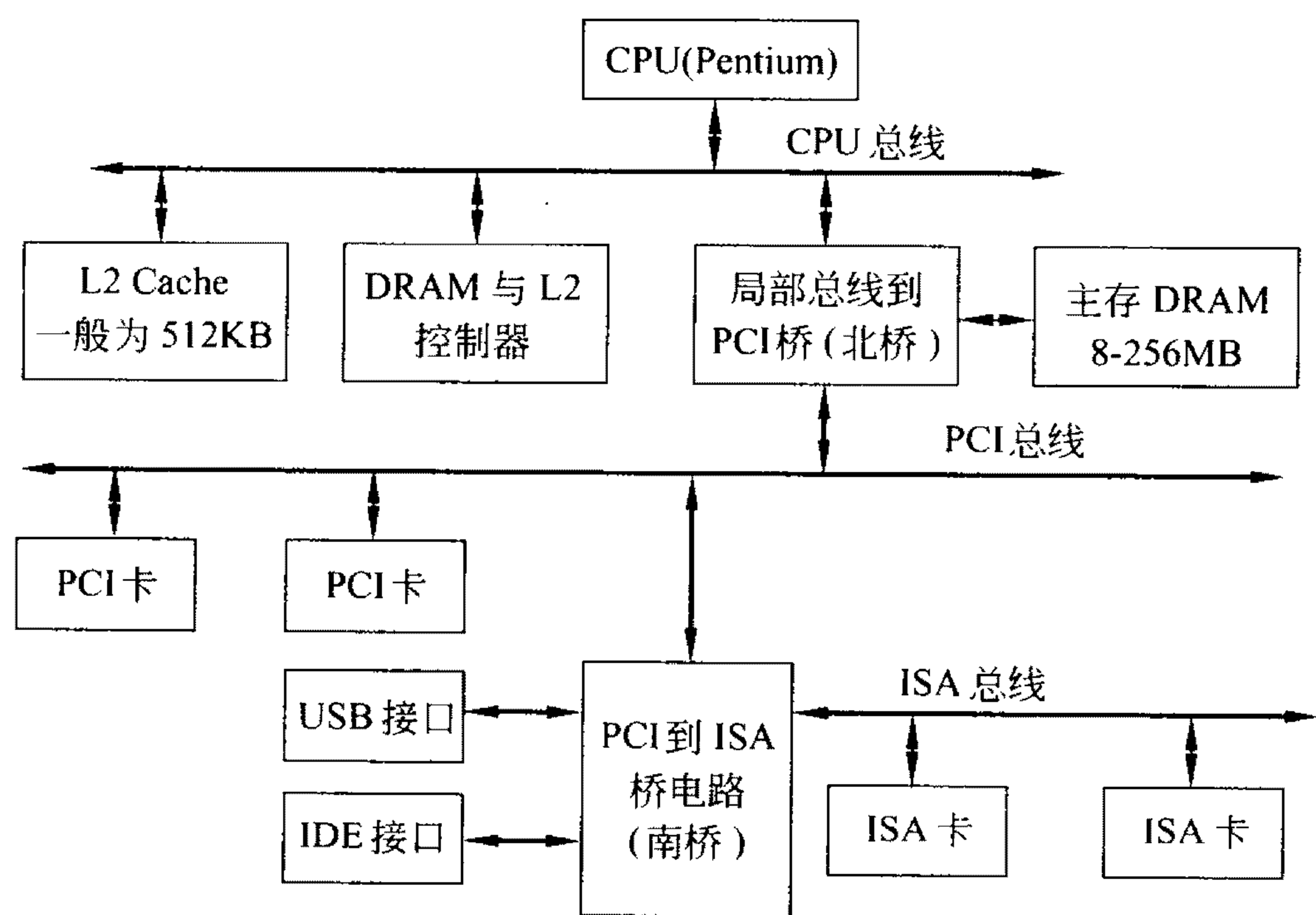


图 6-7 32 位 (Pentium) 微型计算机总线结构

其中北桥芯片与 CPU、内存、二级高速缓存、局部总线等高速设备相连,用来管理微型计算机体系结构中的高速设备;南桥芯片与 IDE、ISA 总线等低速设备相连,用来管理微型计算机体系结构中的低速设备。在兼容 PCI 总线规范的微型计算机技术体系结构中,微处理器局部总线通过一个专门的局部总线连接到 PCI 总线体系结构控制逻辑,与其他部件相连接。每当微处理器及其局部总线改变时,只需跟着改变北桥芯片,全部原有外围设备则可自动继续工作。这种结构的好处就是:即使微处理器及局部总线发生变动,也不会影响众多的外围芯片系列。

图 6-8 所示的 32 位体系结构,是一种中心结构的 Pentium III 微型计算机的体系结构。构成这种结构的芯片组主要由三个芯片组成,分别是存储控制中心 MCH(Memory Controller Hub)、I/O 控制中心 ICH(I/O controller Hub)和固件中心 FWH(Firm Ware Hub)。

MCH 的用途是提供 AGP 接口、动态显示管理、电源管理和内存管理功能。此外, MCH 与 CPU 总线相连,负责处理 CPU 与体系结构其他部件之间的数据交换。在某些类型的芯片组中,MCH 还内置图形显示子体系结构,既可以直接支持图形显示又可以采用 AGP 显示部件,称其为图形存储控制中心(GMCH)。

ICH 含有内置 AC'97 控制器提供音频编码和调制解调器编码接口、IDE 控制器提供高速硬盘接口、2 个或 4 个 USB 接口、局域网络接口以及和 PCI 插卡之间的连接。此外, ICH 和 Super I/O 控制器相连接,而 Super I/O 控制器主要为体系结构中的慢速设备提供与体系结构通信的数据交换接口,比如串行口、并行口、键盘和鼠标等。

固件中心包含了主板 BIOS 和显示 BIOS 以及一个用于数字加密、安全认证等领域的硬件随机数发生器。

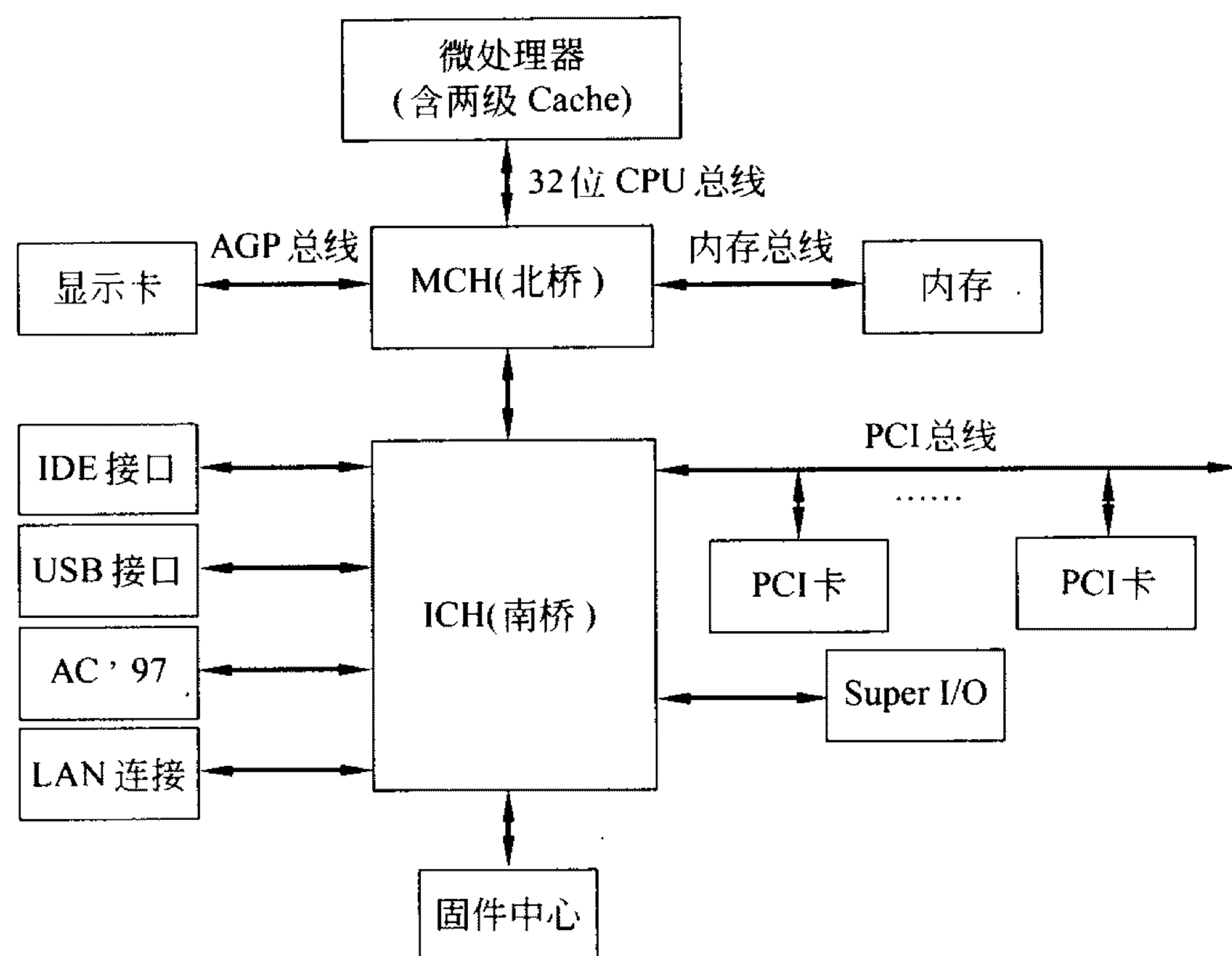


图 6-8 32 位 (Pentium III) 微型计算机总线结构

由上述两种微型计算机的总线结构可以看出：随着计算机 CPU 和芯片组技术的发展，新推出的微型计算机的体系结构和性能可能有些不同。不同公司生产的 CPU 不同，芯片组不同，微型计算机技术的体系结构也有些不同，但其大致结构还是一样的。

习 题

1. 什么是总线？按总线连接的对象和所处系统的层次来分，有哪几类总线？
2. 从哪几方面可以衡量总线的性能？
3. 总线信息的传送方式有哪几种？ISA 总线标准中， \overline{IOW} 、 \overline{IOR} 、 \overline{MEMR} 和 \overline{MEMW} 信号的作用是什么？
4. PCI 总线的特点有哪些？
5. 哪些标准是通用外部总线标准？分别适合什么外部设备使用？
6. USB 的特点是什么？它支持哪几种基本的数据传输模式？
7. 什么是 PCI 桥接器？PCI 规范中提出了几类桥的设计？分别是什么？
8. 什么是 32 位微型计算机系统总线结构中的南桥和北桥？各自有什么作用？

存储器系统

7.1 概 述

目前绝大多数计算机硬件系统仍然是冯·诺依曼“存储程序”式结构。“存储程序”思想的核心是将编好的程序和要加工处理的数据预先存入主存储器(Main Memory),然后启动计算机工作,计算机在不需人工干预的情况下,高速自动地从主存储器中取出指令执行,从而完成数值计算或非数值处理。显然,存储器是实现“存储程序”控制必不可少的硬件支持,是计算机中必须要有的重要组成部分。

存储器是具有记忆功能的部件,是能够接收、保存和取出信息(程序、数据和文件)的设备。

7.1.1 存储系统概念

目前的计算机系统,大都采用多种类型存储器。几乎没有只用单一存储器的情形,原因是要在容量(S)、速度(T)、价格(C)三者之间折衷。速度快的存储器价格贵,容量就不能做得很大,而价格低的存储器容量可能做得相当大,但它的存取速度却比较慢。为解决速度、容量和价格之间的矛盾,人们提出了存储系统的概念。

由 $n(n \geq 2)$ 个速度、容量、价格各不相同的存储器组成由硬件或软件进行辅助管理的系统称为存储系统。图 7-1 是一个典型的存储系统。这个系统对应用程序员透明,并且从应用程序员看它是一个存储器,这个存储器的速度接近速度最快的那个存储器,存储容量与容量最大的那个存储器相等或接近,单位容量的价格接近最便宜的那个存储器。

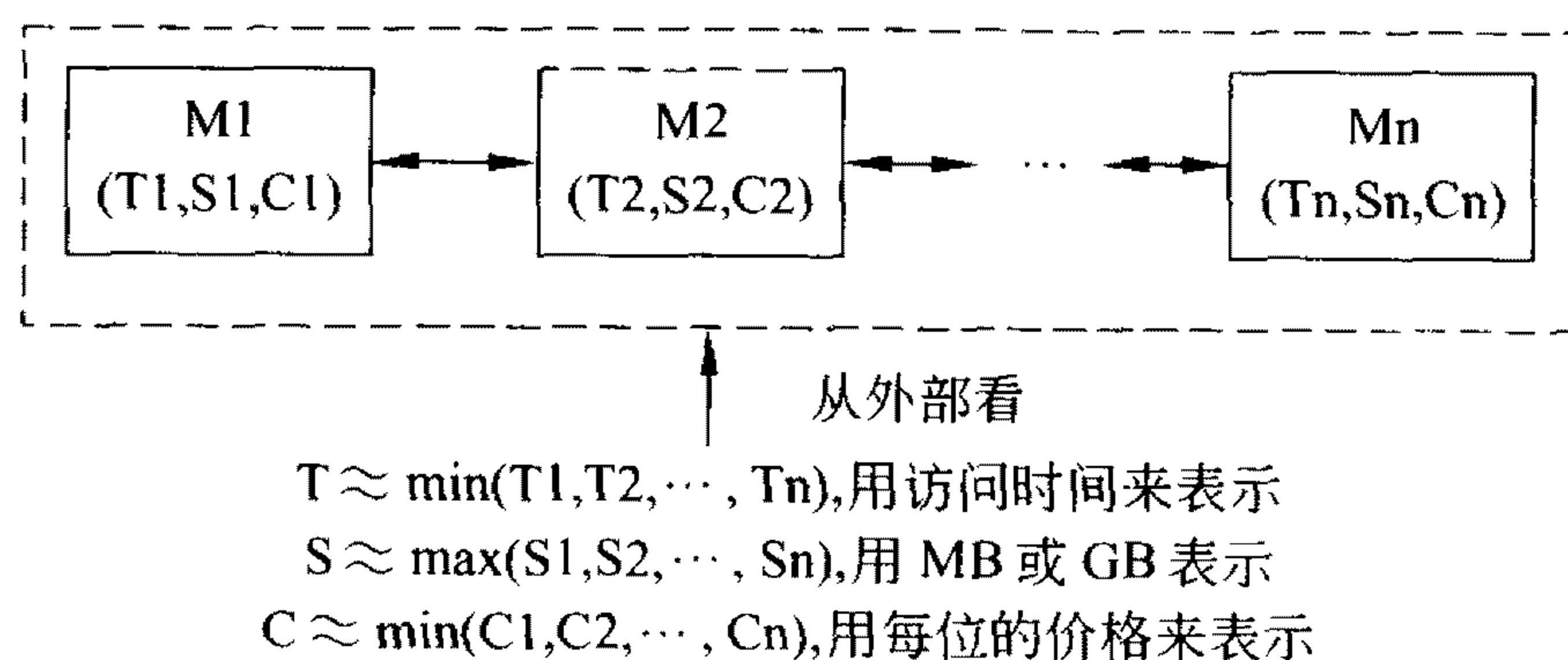


图 7-1 存储系统原理

图中 $M_i(i=1,2,\dots,n)$ 代表存储体 i ; $T_i(i=1,2,\dots,n)$ 代表存储体 i 的速度,用访问时间来表示; $S_i(i=1,2,\dots,n)$ 代表存储体 i 的容量,用 MB 或 GB 表示; $C_i(i=1,2,\dots,n)$ 代表存储体 i 的价格,用每位的价格来表示。

存储系统概念早在 20 世纪 60 年代就运用于计算机系统之中。随着集成电路技术的飞速发展,许多大型计算机甚至巨型计算机的成熟技术已经逐步下移至微型计算机。存储系统就是其中一项主要的技术。

7.1.2 存储器的体系结构

存储系统的设计始终是围绕着解决速度(访问时间 T)、容量(S)和价格(C)之间的矛盾而进行的。人们在进行存储体系的设计时,特别是在大型计算机系统的系统结构中,建立了存储层次体系结构的概念。所谓存储层次体系结构(memory hierarchy)是在综合考虑容量、速度、价格的基础上,建立的存储组合,以满足系统对存储器在性能与经济两个方面的要求。

1. 访存局部性原理

硬件系统中的一个普遍的规律是:系统的复杂性越小,系统能达到的速度越高,系统的尺寸越小,系统能达到的速率越高。对于存储器系统,高速、大容量和低成本这三个因素是相互矛盾的。尽管在大多数计算机应用中往往需要巨大的存储空间,但程序对其存储空间的访问并不是均匀分布的。从大量的统计中可以得到这样一个规律:程序对存储空间的 90% 的访问局限于存储空间的 10% 的区域中,而另外 10% 的访问则分布在存储空间的其余 90% 的区域中。这一规律就是通常所说的访存局部性原理。访存局部性原理包含两个方面:

- ① 时间局部性:如果一个存储项被访问,则该项可能很快会被再次访问。
- ② 空间局部性:如果一个存储项被访问,则该项及相邻近的项也可能很快被访问。

形成上述规律的原因在于程序的顺序执行和程序的循环等。在程序顺序执行时,下一次执行的指令和上一次执行的指令在存储器中的位置是相邻的或相近的;在循环程序中,循环体的指令要重复执行,相应的数据就要重复访问。指令和数据的存放都是有一定的规律而不是随机的。

2. 层次化存储系统

根据访存局部性原理来解决存储器容量和速度的矛盾,就是要求将计算机频繁访问的数据存放在速度较高的存储介质中,而将不频繁访问的数据存放在速度较慢但价格较低的存储介质中,为此人们想到了层次化的存储器实现方法。存储器系统根据容量和工作速度分成若干个层次,因为速度较慢的存储介质成本较低,用其实现较低层次的存储器,而用少量的速度较高的存储器件实现速度要求较高的存储层次。在多个层次的存储器中,上一层次的存储器较下一层次的容量小,速度快,每字节的成本更高,距离处理机更近。访问频率高的数据存放在层次高的存储器中并在其下层存储器有一原始备份,这样既可以用较低的成本实现大容量的存储器,又使存储器具有较高的平均访问速度,

图 7-2表示了按这种方式构成的存储器系统。

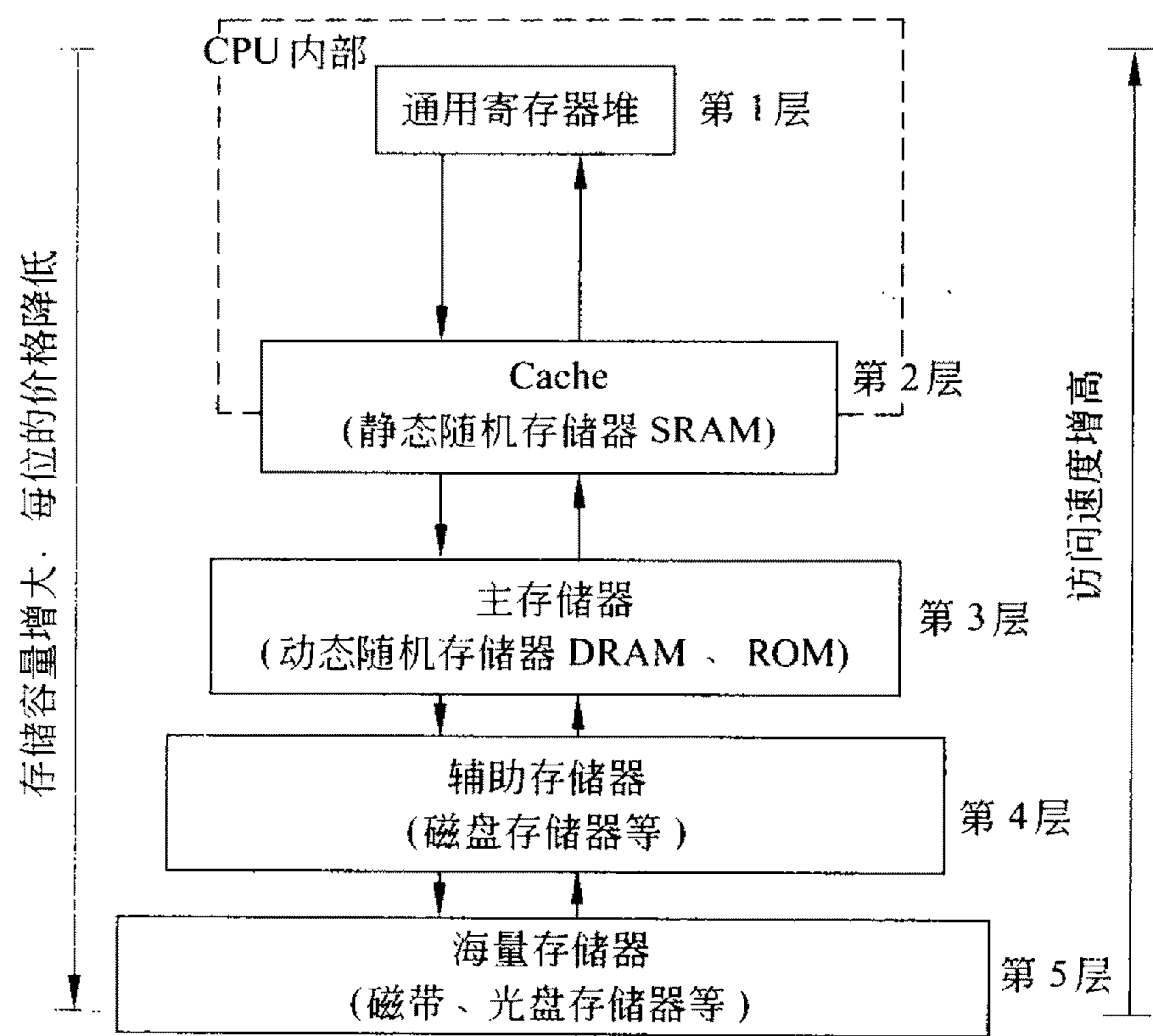


图 7-2 层次化存储器体系结构

CPU 中的寄存器可看作是最高层次的存储部件，它容量最小，速度最快，但寄存器对程序员是不透明的，对寄存器的访问不是按地址而是按寄存器名。寄存器以下可以有指令和数据缓冲栈、高速缓存、主存、辅存和海量存储器等层次。海量存储器是最低层次的存储器，通常由磁盘或磁带构成，它的容量大，成本低但存取速度慢。

3. 存储器系统的设计目标

近几年 CPU 的设计水平不断提高，其速度成倍增加，使存储器成为计算机系统明显的性能瓶颈。尽管近年来存储技术的发展很快，但存储器器件还是无法以合理的成本满足这一要求。速度、容量和价格是存储器系统设计应考虑的三个主要因素。存储器系统设计目标之一就是要以较小的成本使存储器系统与处理机的速度相匹配，或者说达到与处理机相应的工作速度和传输频带宽度。同时还要求存储器有尽可能大的容量。

例如，图 7-2 构成的典型大型计算机的存储器系统特性如表 7-1 所示。

表 7-1 典型大型计算机的存储器系统特性

存储器系统层次特性	第 1 层 CPU 寄存器	第 2 层 高速缓存	第 3 层 主存储器	第 4 层 磁盘存储器	第 5 层 磁带存储器
访问时间	10ns	25ns	60ns	10ms	2min
容量(KB)	512B	128KB	512MB	60GB	2TB
成本(美分/字节)	18000	72	5.6	0.23	0.01

从表 7-1 可以得出该存储器系统的每 KB 的价格为：

$$\frac{\sum_{i=1}^5 C_i S_i}{\sum_{i=1}^5 S_i} = \frac{18000 \times 0.5 + 72 \times 128 + 5.6 \times 512 \times 10^3 + 0.23 \times 60 \times 10^6 + 0.01 \times 2 \times 10^9}{0.5 + 128 + 512 \times 10^3 + 60 \times 10^6 + 2 \times 10^9}$$

$$= 0.0178(\text{美分})$$

该存储器系统的存储容量不小于 2TB, 平均访问时间几乎不大于 25ns。由此可见, 该存储系统的容量相当于第 5 层, 价格比第 5 层略高, 速度相当于第 2 层, 所以该存储系统是一个成功的设计范例。

7.1.3 存储器的分类

构成存储器的存储介质, 目前主要采用半导体器件和磁性材料。一个双稳态半导体电路或磁性材料的存储元, 均可以存储一位二进制代码。这个二进制代码位是存储器中最小的存储单位, 称为一个存储位或存储元。由若干个存储元组成一个存储单元, 然后再由许多存储单元组成一个存储器。

根据存储元件的性能及使用方法不同, 存储器有各种不同的分类方法。

1. 按存储介质分类

存储介质必须能够显示两个有明显区别的物理状态, 分别用来表示二进制代码的 0 和 1。另一方面, 存储器的存取速度又取决于这种物理状态的改变速度。目前使用的存储介质主要是半导体器件和磁性材料。用半导体器件组成的存储器称为半导体存储器。用磁性材料做成的存储器称为磁表面存储器, 例如磁盘存储器和磁带存储器。

2. 按存取方式分类

如果存储器中任何存储单元的内容都能被随机存取, 且存取时间和存储单元的物理位置无关, 这种存储器称为随机存储器。半导体存储器和磁芯存储器都是随机存储器。如果存储器只能按某种顺序来存取, 也就是说存取时间和存储单元的物理位置有关, 这种存储器称为顺序存储器。例如, 磁带存储器就是顺序存储器。一般来说, 顺序存储器的存取周期较长。磁盘存储器是半顺序存储器。

3. 按存储器的读写功能分类

有些半导体存储器存储的内容是固定不变的, 即只能读出而不能写入, 因此这种半导体存储器称为只读存储器(ROM)。既能读出又能写入的半导体存储器, 称为随机存储器(RAM)。

4. 按信息的可保存性分类

断电后信息即消失的存储器, 称为非永久记忆的存储器。断电后仍能保存信息的存储器, 称为永久性记忆的存储器。磁性材料做成的存储器是永久性记忆的存储器。半导体读写存储器 RAM 是非永久性记忆的存储器。

5. 按串、并行存取方式分类

目前使用的半导体存储器大多为并行存取方式,但也有以串行存取方式工作的存储器如电耦合器件(CCD)、串行移位寄存器和镍延迟线构成的存储器等。

6. 按在计算机系统中的作用分类

根据存储器在计算机系统中所起的作用,可分为主存储器、辅助存储器、缓冲存储器和控制存储器等。如图 7-3 所示。

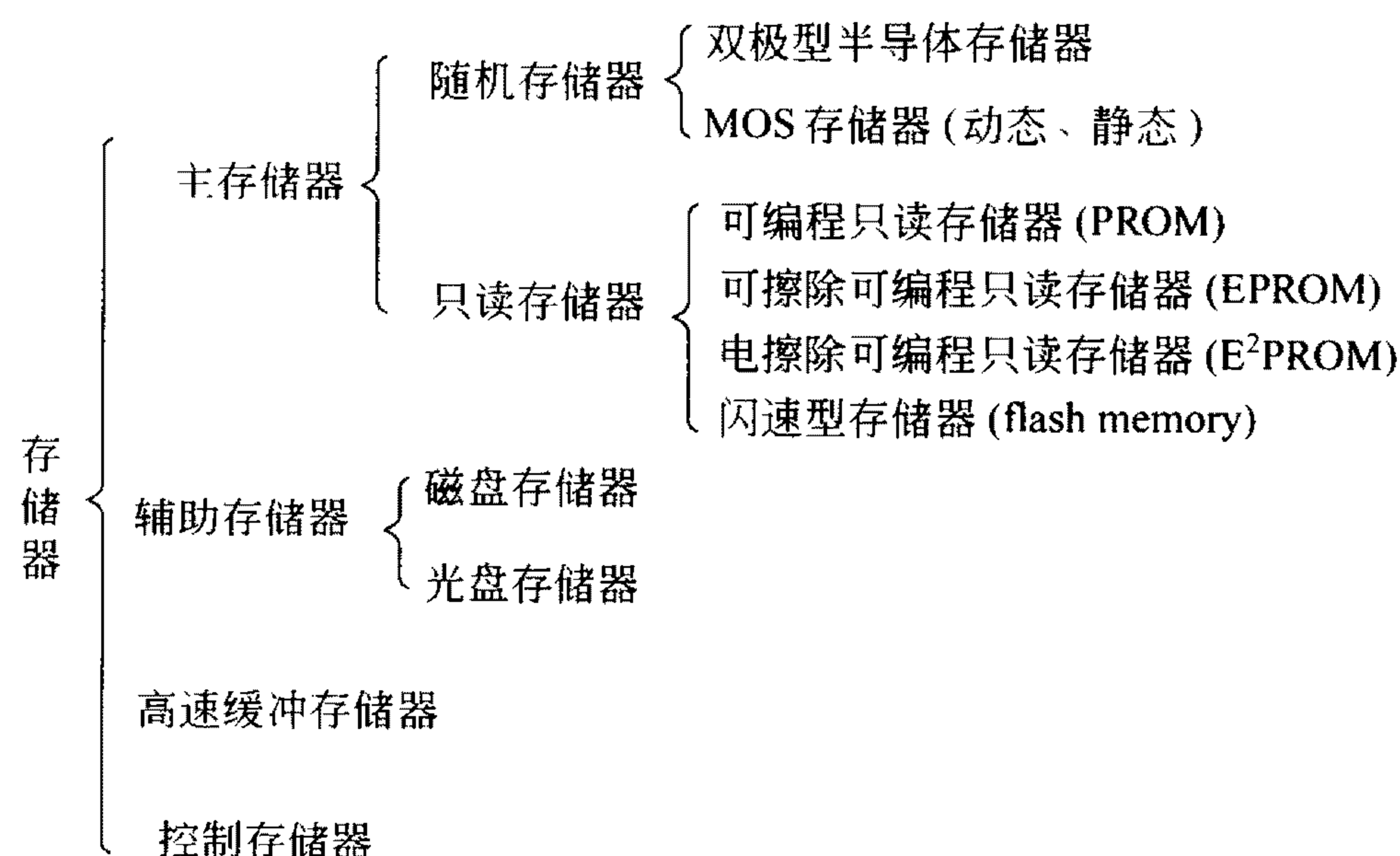


图 7-3 计算机系统中存储器分类

7.1.4 存储器的主要性能指标

存储器的类型不同,其性能指标也不相同,在构成微型计算机硬件系统时需要全面考虑。存储器的主要性能指标有以下几点:

1. 存储器容量(Memory Capacity)

在微型计算机中,存储器以字节为单元。每个单元包含 8 位二进制数,也就是一个字节。由于存储容量一般都很大,因此常以 KB(2^{10} 字节)、MB(2^{20} 字节)或 GB(2^{30} 字节)为单位。目前高档微型计算机的内存容量一般为 32MB ~ 4GB。存储器容量越大,存储的信息量也就越大,计算机运行的速度也就越快。

存储一位二进制信息的单元称为一个基本单元。对于 32MB 的存储器,其内部有 $32\text{M} \times 8\text{bit}$ 个基本单元。

2. 存取时间(Memory Access Time)

存取时间也称为存储器访问时间,指的是从启动一次存储器操作到完成该操作所用的时间。如从发出读命令到将数据送入数据缓冲寄存器所用的时间,或从发出写命令到将数据缓冲寄存器的内容写入相应存储单元所用的时间,用 T_A 表示。 T_A 是反映存储器



速度的指标,其值取决于存储介质的物理特性及其使用的读出机构的类型。 T_A 决定了CPU进行一次读或写操作必须等待的时间。目前主存的存取时间为纳秒级。

3. 存储周期 (Memory Cycle Time)

存储周期也称为存取周期、访问周期、读写周期。指的是连续两次启动同一存储器进行存储操作所需的最小时间间隔,用 T_M 表示。因为对任一种存储器,当进行一次访问后,存储介质和有关控制线路都需要恢复时间,若是破坏性读出,还需重写时间,因此通常 $T_M > T_A$ 。 T_A 通常主要用来表示 CPU 发出读命令后要等待多长时间才能获得数据,这对 CPU 的设计有着非常重要的意义。如果考虑计算机与访存有关的工作周期,则会涉及 T_M 。

4. 可靠性 (Reliability)

计算机的一切工作都是通过运行程序实现的,而正在运行的程序和要加工处理的数据都存放在存储器中,因此,存储器的可靠性处于非常重要的地位。通常用平均无故障时间 MTBF (Mean Time Between Failures) 来衡量存储器的可靠性,MTBF 表示两次故障之间的平均时间间隔。显然,MTBF 越大,可靠性越高。为了加大 MTBF,存储器采用容错技术。所谓容错,就是在存储器出现故障时,能够纠正错误,使之正常工作,或者至少能报告错误,以便人工排除。通常通过增加冗余位实现,如 YH-2 巨型机,CPU 字长 64 位,而存储器字长为 72 位,多用 8 位可纠正一位错误、检测出两位错误,这样大大提高了存储器的可靠性。

5. 功耗与集成度 (Power Loss and Integration Level)

功耗反映了存储器件耗电多少,集成度标识单个存储芯片的存储容量。一般希望功耗小、集成度高,但两者是矛盾的,因此除设计和制作存储芯片时要同时考虑两者之外,用芯片构成存储器时也应考虑它们。对于存储器件,有维持功耗和工作功耗之分,工作功耗远远大于维持功耗,通常要求维持功耗尽量小。

对于高密度组装的高速存储器,则应采用风冷、液冷等强化散热措施,否则存储器将不会稳定工作,甚至有烧毁的危险。

6. 性能价格比 (Cost Performance)

性能价格比是一个综合性指标,性能主要包括存储容量、存储周期、存取时间和可靠性等。价格包括存储芯片和外围电路的成本。通常要求性能价格比要高。

7. 存取宽度 (Access Width)

存取宽度也称为存储总线宽度,即 CPU 或 I/O 一次访存可存取的数据位数或字节数。存取宽度由编址方式决定,字节编址存取宽度为 8 位,字编址存取宽度为机器的字长,它一般是字节的整倍数。如银河-I 巨型机存取宽度为 64 位。低档微型计算机存取宽度为 8 位、16 位。高档微型计算机存取宽度为 32 位、64 位。

7.2 随机存储器与只读存储器

计算机系统的存储器部分通常由只读存储器(ROM)和随机访问存储器(RAM)构成。ROM 用于存储永久性的信息,如计算机的接口驱动程序或数据。RAM 是随机访问存储器。

7.2.1 RAM 的分类与常用 RAM 芯片的工作原理

RAM 是随机访问存储器。随机访问是指不管存储器地址如何,读、写存储单元所需的时间都是相等的;RAM 的另一特点是断电之后内容将丢失。

存放一个二进制位的物理器件称为记忆单元,它是存储器的最基本构件,可以由各种材料制成,MOS 型存储器根据记忆单元的结构又可分为静态 SRAM(Static RAM)和动态 DRAM(Dynamic RAM)两种。SRAM 存储电路以双稳态触发器为基础,其一位存储单元类似于 D 锁存器。数据一经写入,只要不关掉电源,则将一直保持有效。DRAM 存储电路以电容为基础,靠芯片内部电容电荷的有无来表示信息,为了防止由于电容的漏电所引起的信息的丢失,就需要在一定的时间间隔内对电容进行充电,这种充电的过程称为 DRAM 的刷新。对于这种类型的电路,必须既要维持其电源电压,又要定期维持每个存储单元中的数据。

1. 静态 RAM(SRAM)

(1) 基本存储单元电路

SRAM 的基本存储单元电路如图 7-4 所示,它是在 MOS 型触发器的基础上增添了两个门控管。图中 $T_1 \sim T_4$ 构成双稳态触发器,两个稳定状态分别表示 1 或者 0。例如 A 点为高电平,B 点为低电平,表示存 1,相反则表示存 0。 T_5 、 T_6 为门控管,当 X 选择线为

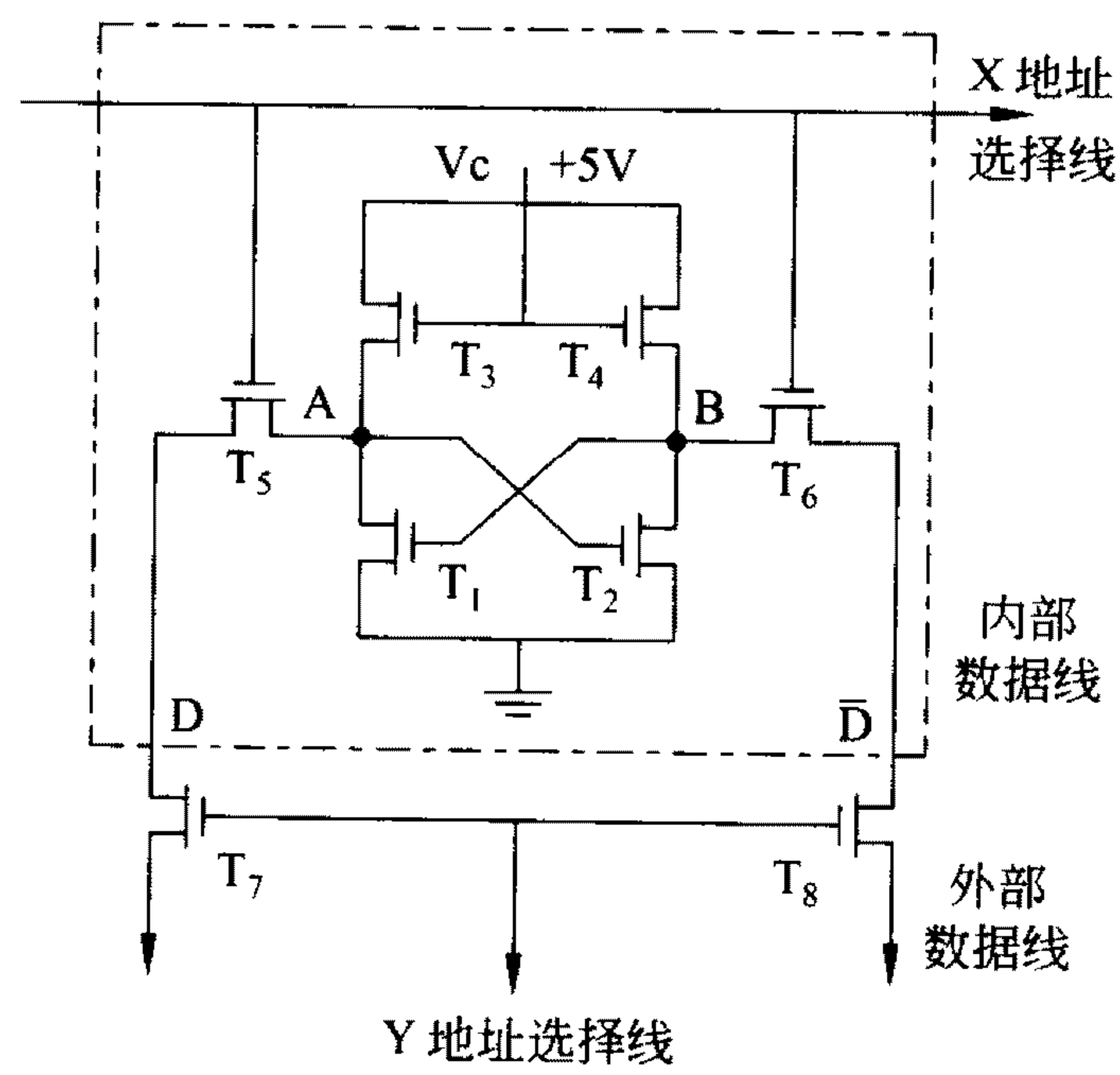


图 7-4 MOS 型 6 管静态存储器

高电平时, T_5 、 T_6 管导通, 表示该单元选中, A 点和 B 点分别与内部数据线 D 和 \bar{D} (也称位线) 接通。 T_7 和 T_8 也是门控管, 控制该存储单元的内部数据线是否与外部数据线接通。当 Y 选择线也为高电平时, T_7 和 T_8 管导通, 内部数据线与外部数据线接通, 表示该单元的数据可以读出, 或者把外部数据线上的数据写入到该存储单元。

在读出时, X 选择线与 Y 选择线均为高电平, T_5 、 T_6 、 T_7 、 T_8 管均导通, A 点与 D 接通, B 点与 \bar{D} 接通, D、 \bar{D} 又与外部数据线接通。若原来存入的是 1, A 点为高电平, 则 D 为高电平; B 点为低电平, 则 \bar{D} 为低电平。二者分别通过 T_7 、 T_8 管输出到外部数据线, 即读出 1。相反, 若 A 点为低电平, 则 D 为低电平; B 点为高电平, 则 \bar{D} 为高电平, 二者分别通过 T_7 、 T_8 管输出到外部数据线, 即读出 0。

在写入时, 首先将要写入的数据送到外部数据线上。若该单元被选中, 则 X 选择线与 Y 选择线为高电平, T_5 、 T_6 、 T_7 、 T_8 管均导通, 外部数据线上的数据就分别通过 T_7 、 T_5 管和 T_8 、 T_6 管送到触发器的 A 点与 B 点。若写入的是 1, 则 T_2 导通, B 点为低电平, T_1 截止, A 点为高电平。写入结束, 状态保持。若写入的是 0, 则状态相反, A 点为低电平, B 点为高电平。

(2) 地址译码方式

地址译码就是选择某一存储单元, 常用的方式有两种: 一种是字译码方式, 也称为单译码方式; 另一种是复合译码方式, 也称为双译码方式。

• 字译码方式

字译码方式如图 7-5 所示, 由一条字选择线一次选择某一字的所有位。在图 7-5 中, 有 16 个存储单元, 分别由 16 条字选择线选择。地址有效后经译码, 使某一字线为高电平, 于是该单元中所有位的门控管导通, 各位数据可通过各自的读/写控制电路读出, 或者将外部数据写入。读/写控制电路受 CPU 发来的读命令和写命令的控制。只有当

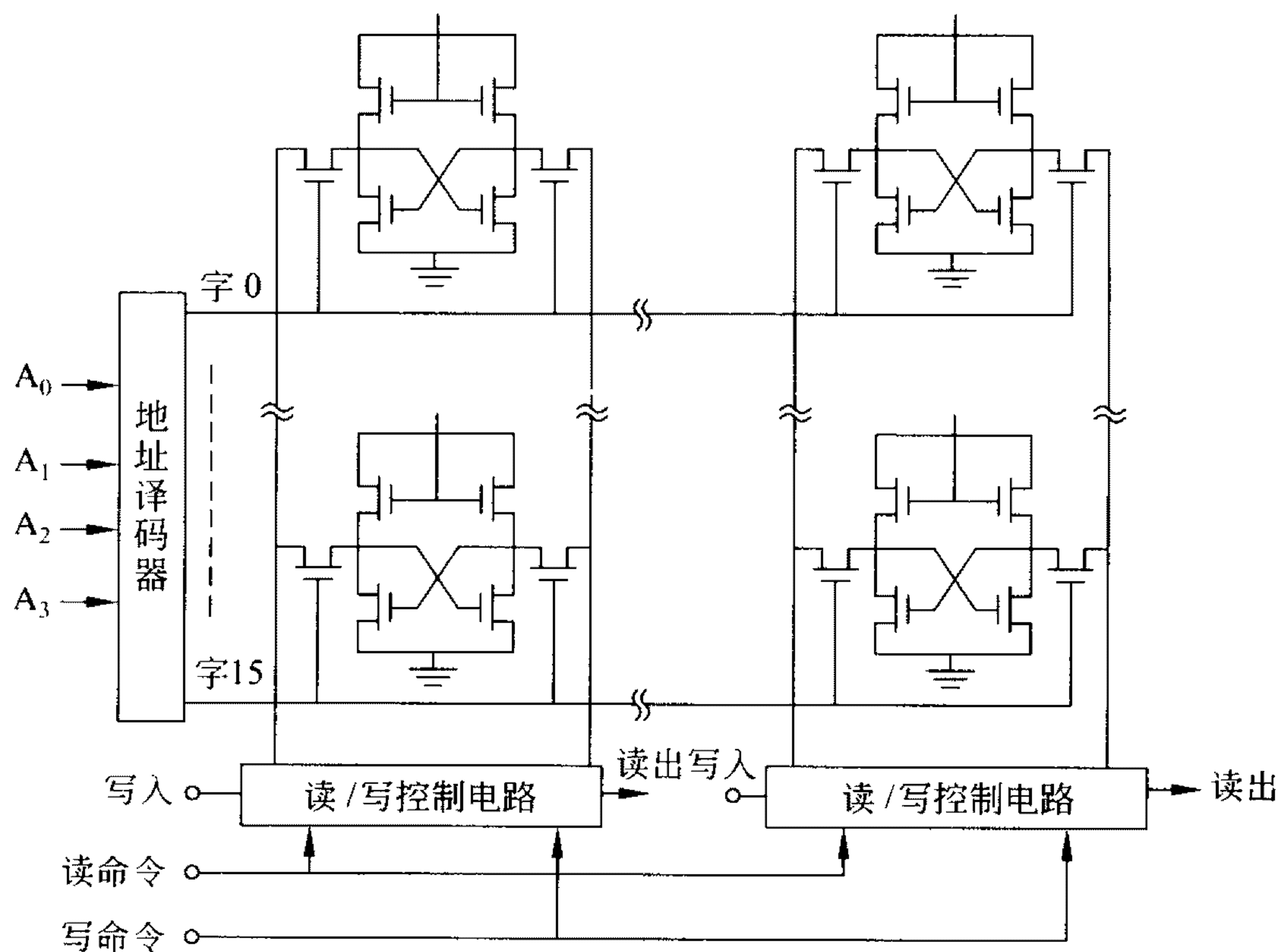


图 7-5 字译码方式存储器

CPU 发出读/写命令后才能对选中的字单元进行相应的读或者写操作。

• 复合译码方式

复合译码方式如图 7-6 所示,由纵横交错的 X 选择线和 Y 选择线互相配合来选择某一存储单元。在这种电路中,基本存储单元排列成阵列(32×32),作为所有单元的同位。对于 8 位存储器,须由 8 个这样的阵列重叠起来组成。它有两个地址译码器,一个是水平方向的 X 地址译码器,它决定选择 32 行中的某一行;另一个是竖直方向的 Y 地址译码器,它决定选择 32 列中的某一列。它的地址分为两部分,一部分 $A_4 \sim A_0$ 送 X 地址译码器;另一部分 $A_9 \sim A_5$ 送 Y 地址译码器。工作时,10 位地址分为两部分,分别由 X、Y 地址译码器译码,选择出某一行和某一列交叉处的一个存储单元。

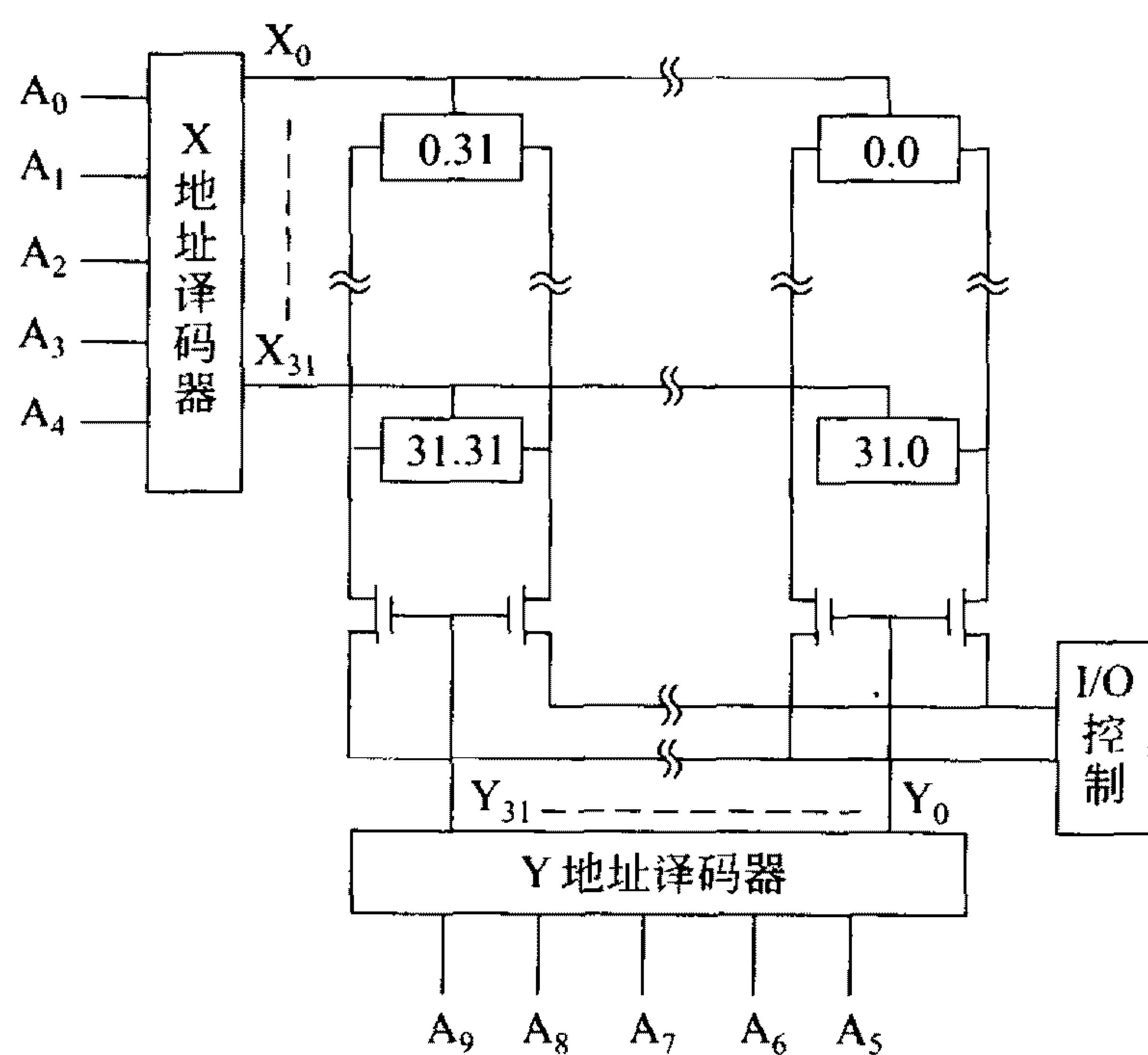


图 7-6 符合译码方式存储器

字译码方式常用于小容量存储器,对于大容量的存储器多采用复合译码方式,或者把二者结合起来使用。

(3) 静态随机存取存储器实例

在构成存储器时,一般以字节为单位。目前生产的存储器芯片有 1 位片、4 位片和 8 位片等。1 位片是指一个地址单元仅存放 1 位二进制代码。4 位片和 8 位片是指一个地址单元分别存放 4 位或 8 位二进制代码。这样在构成以字节为单位的存储器时,若采用 1 位片或 4 位片,就需要 8 片或 2 片连接使用,即构成一个 8 位的芯片组。如果要构成一个大容量的存储器,就需要多个芯片组连接起来。当然,若直接采用 8 位芯片,如 Intel 2128、6116、2186、6264、62128、62256、62512 等,则最为方便。

• 6264 的基本组成与特点

6264 是一种 $8K \times 8\text{bit}$ 的静态存储器。其内部组成如图 7-7(a)所示,主要包括 256 行×256 列的存储器矩阵、行/列地址译码器以及数据输入输出控制逻辑电路。地址线 13 位,其中 $A_{12} \sim A_3$ 用于行地址译码, $A_2 \sim A_0$ 和 A_{10} 用于列地址译码。在存储器读周期。选中单元的 8 位数据经列 I/O 控制电路输出;在存储器写周期,外部 8 位数据经

输入数据控制电路和列 I/O 控制电路,写入到所选中的单元中。6264 有 28 个引脚,如图 7-7(b)所示,采用双列直插式结构,使用单一+5V 电源。其引脚功能如下:

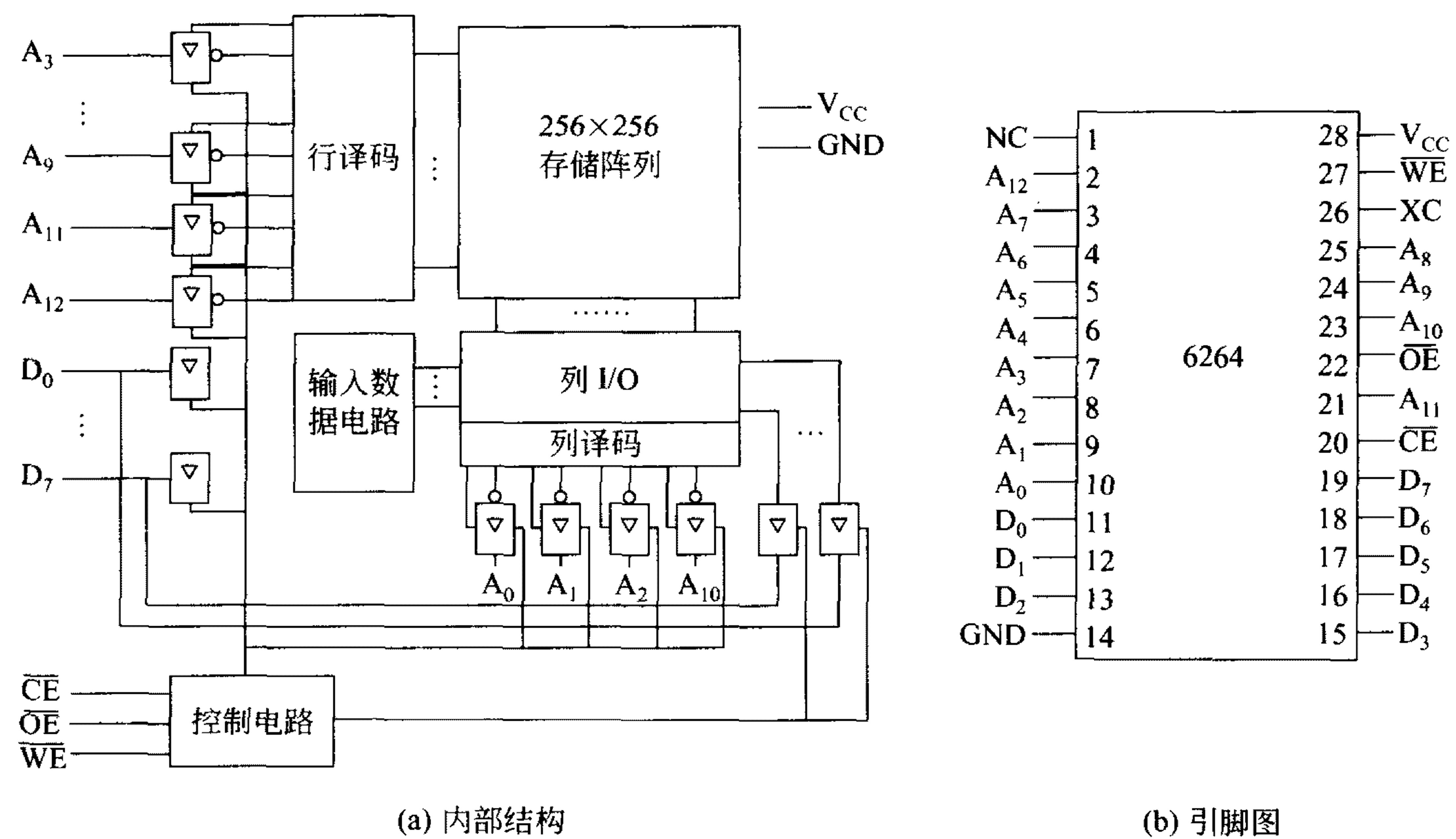


图 7-7 6264 内部结构和引脚图

- $A_{12} \sim A_0$: 地址线,输入,寻址范围为 8KB;
 - $D_7 \sim D_0$: 数据线,8 位,双向传送数据。
 - \overline{CE} : 片选信号,输入,低电平有效。
 - \overline{WE} : 写允许信号,输入,低电平有效。
 - \overline{OE} : 读允许信号,输入,低电平有效。
 - V_{CC} : +5V 电源。
 - GND: 接地。
 - NC: 未用。
- 6264 的工作方式如表 7-2 所示。

表 7-2 6264 工作方式选择

\overline{CE}	\overline{WE}	\overline{OE}	方 式	功 能
0	0	0	禁止	不允许 \overline{WE} 和 \overline{OE} 同时为低电平
0	1	0	读出	数据读出
0	0	1	写入	数据写入
0	1	1	选通	芯片选通,输出高阻态
1	X	X	未选通	芯片未选通

2. 动态随机存取存储器 DRAM

(1) 基本存储单元电路

在六管静态存储器中,信息的存储在于 T_1 、 T_2 管的导通和截止,而 T_1 、 T_2 管的导通和截止靠负载管 T_3 、 T_4 来维持。对于 MOS 管,它有一个显著的特点,就是栅电容、栅电阻比较大,栅漏电流很小。这样栅电容可以暂存一定数量的电荷,维持 MOS 管导通片刻时间。如果不断地给栅电容补充电荷,则可以维持 MOS 管一直导通下去。为了减少单元电路的晶体管数,提高集成度,可以把负载管 T_3 、 T_4 取掉。这样就变成四管动态存储器,如图 7-8 所示。在图 7-8 中 T_5 、 T_6 、 T_7 、 T_8 管仍为门控管, T_9 、 T_{10} 为预充电管。

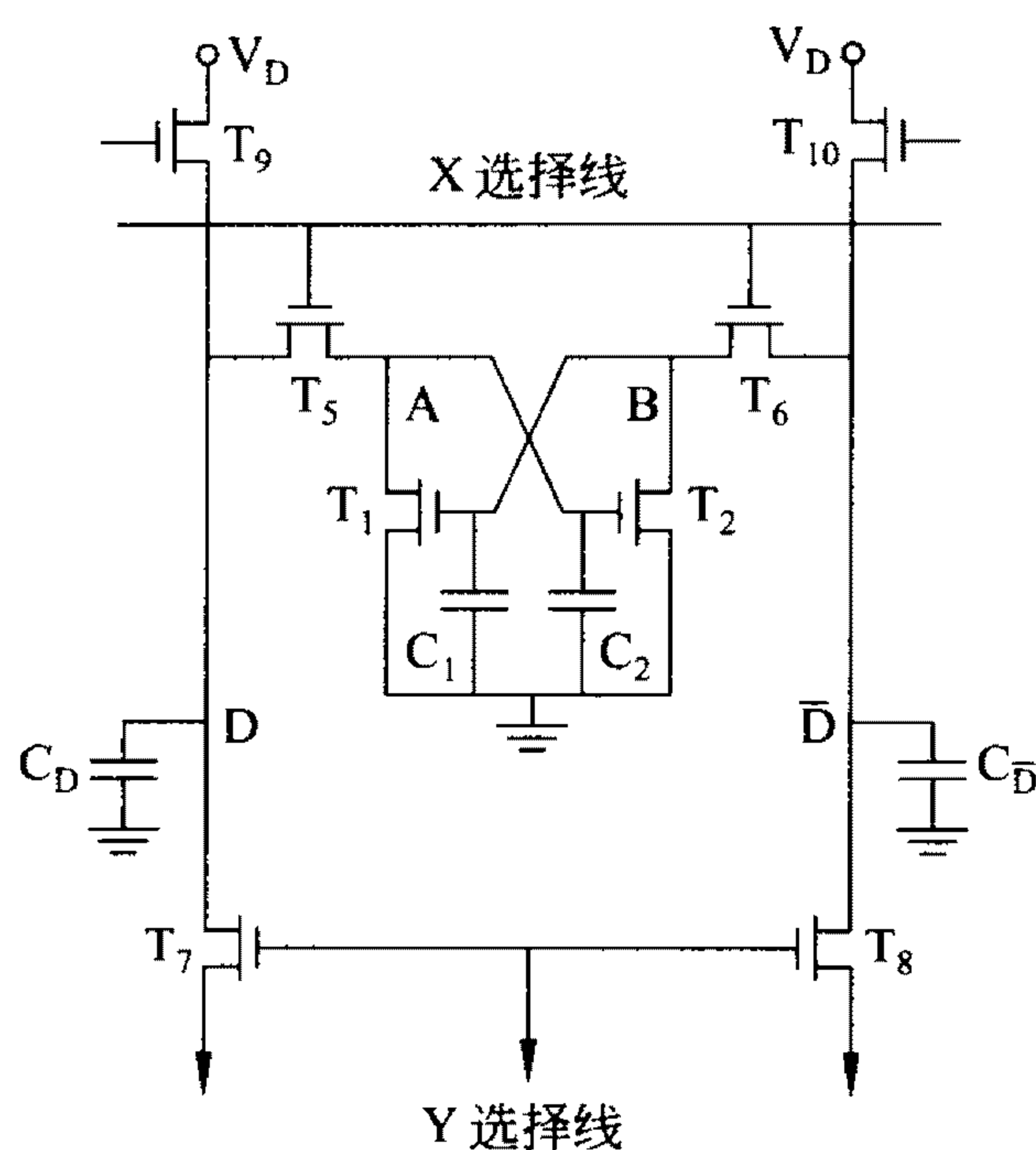


图 7-8 MOS 型四管动态存储器

写入时, X 、 Y 选择线为高电平, T_5 、 T_6 、 T_7 、 T_8 管导通, 该单元被选中, 外部数据线上的数据分别通过 T_7 、 T_8 管送到 D 和 \bar{D} 。再通过 T_5 、 T_6 管达到 A 点和 B 点。当写入 1 时, A 点送入的是高电平, 于是栅电容 C_2 充电, T_2 导通, B 点送入的是低电平, 栅电容 C_1 不充电, T_1 截止。写入结束, 1 状态依靠 C_2 存储的电荷维持一段时间。为了使存储的信息能长时间的保持下去, 就要每隔一定的时间给 C_2 补充电荷, 这一工作通常称为动态存储器的刷新或者再生。写入 0 时, 与之相反, C_1 充电, T_1 导通, A 点为低电平, C_2 不充电, T_2 截止, B 点为高电平。

读出时, 由于栅电容上存储的电荷很少, 不能产生一个大的读出信号, 因此在读出时要采取一定的措施, 给电路补充一定的电量, 通常通过预充电的办法来解决。首先发预充电信号, 使预充电管 T_9 、 T_{10} 导通, 由电源 V_D 对内部数据线 D 和 \bar{D} 预充电。当 X 、 Y 选择线为高电平时, 该单元被选中。若原存 1, 即 C_2 充电, T_2 管导通, C_1 不充电, T_1 管截止。由于 X 选择线为高电平, T_5 、 T_6 管导通, 则 \bar{D} 数据线上的高电平经 T_6 、 T_2 管放掉, D 数据线上的高电平保持, 然后再经 T_7 、 T_8 管输出到外部数据线上, 即读出 1。若原存 0, 则与之相反, C_1 充电, T_1 管导通, C_2 不充电, T_2 管截止。读出时, D 数据线输出低电平, \bar{D} 数据线输出高电平, 即读出 0。

在读出过程中, 没有放电的数据线 D 或者 \bar{D} 给 T_1 或者 T_2 管的栅极充电, 补充因栅漏电流而损失的电荷, 因此读出的过程也是动态存储器刷新或者再生的过程。动态存储器的刷新实质上就是一次读操作, 只是读出的信息丢掉不用就是了。

动态存储器除了上述四管电路之外, 还有三管和单管电路。

(2) 动态随机存取存储器举例

由于动态存储器电路简单, 因此相对集成度要比静态存储器高得多, 并且常作为微型计算机的主存储器。目前常用的有 4164、41256、41464 以及 414256 等类型, 其存储容量分别为 $64K \times 1\text{bit}$ 、 $256K \times 1\text{bit}$ 、 $64K \times 4\text{bit}$ 和 $256K \times 4\text{bit}$ 。其中 414256 的内部组成如图 7-9 所示。

414256 的基本组成是 $512 \times 512 \times 4$ 的存储器阵列。在此基础上设有读出放大器与

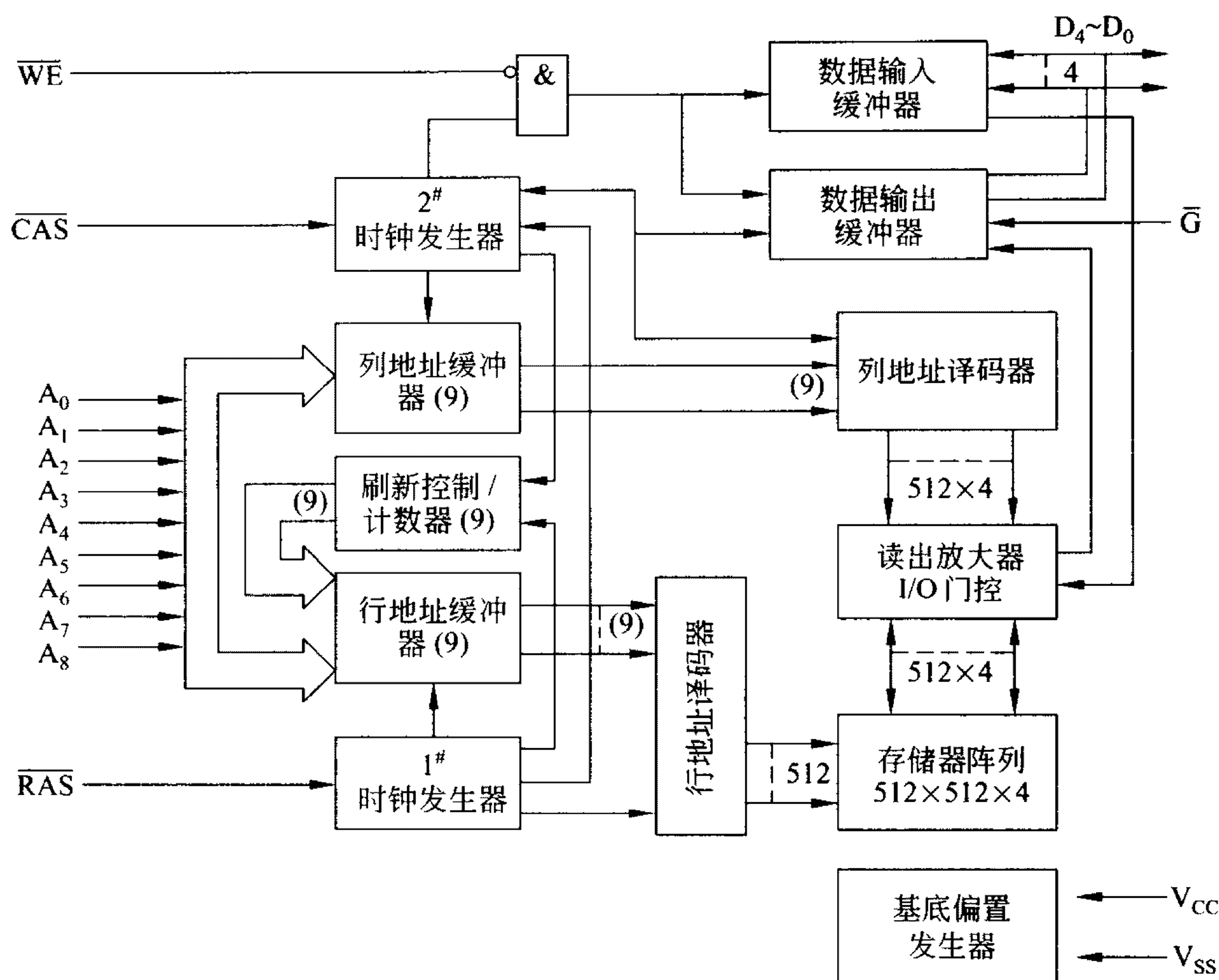


图 7-9 414256/41LA256 内部组成

I/O 门控制电路、行地址缓冲器/译码器、列地址缓冲器/译码器、数据输入/输出缓冲器、刷新控制/计数器以及时钟发生器等。存储器访问时,行地址和列地址分两次输入。首先由 $\overline{\text{RAS}}$ 信号锁存由地址线 $A_8 \sim A_0$ 输入的 9 位行地址,然后再由 $\overline{\text{CAS}}$ 信号锁存由地址线 $A_8 \sim A_0$ 输入的 9 位列地址,经译码选中某一存储单元。在读/写控制信号 $\overline{\text{WE}}$ 的控制下,可对该单元的 4 位数据进行读出或者写入。

由于动态存储器读出时需预充电,因此每次读/写操作均可进行一次刷新。MCM414256 需要每 8ms 刷新一次。刷新时通过在 512 个行地址间按顺序循环进行刷新,可以分散刷新,也可以连续刷新。分散刷新也称为分布刷新,是指每 $15.6\mu\text{s}$ 刷新一行;连续刷新也称猝发方式刷新,它是对 512 行集中刷新。MCM414256 必须每 8ms 进行一次快速刷新,MCM41L4256 每 64ms 进行一次快速刷新。刷新方式有以下三种:

- $\overline{\text{RAS}}$ 刷新

刷新时只产生 $\overline{\text{RAS}}$,锁存行地址; $\overline{\text{CAS}}$ 为高电平。为了确保在一定范围内对所有行都刷新,使用一种外部计数器。

- $\overline{\text{CAS}}$ 在 $\overline{\text{RAS}}$ 之前的刷新

这种方式是在 $\overline{\text{RAS}}$ 之前使 $\overline{\text{CAS}}$ 有效,启动内部刷新计数器,产生需要刷新的行地址,而忽略外部地址线上的信号。

- 隐式刷新

隐式刷新是在数据输入/输出有效时启动一次 $\overline{\text{CAS}}$ 在 $\overline{\text{RAS}}$ 之前的刷新。它是在读/

写周期的末尾保持 $\overline{\text{CAS}}$ 有效,使 $\overline{\text{RAS}}$ 无效,接着再使 $\overline{\text{RAS}}$ 有效,即按 $\overline{\text{CAS}}$ 在 $\overline{\text{RAS}}$ 之前的方式刷新。

7.2.2 ROM 的分类与常用 ROM 芯片的工作原理

半导体只读存储器 ROM 中的内容是事先编制好的。使用时,只是从 ROM 中读出数据。ROM 也具有按地址随机访问的特点;另外,即使断电,它的内容也不会丢失。

ROM 被广泛地应用于微型计算机中,用于存储确定整个系统操作的程序或关键性的常数。存储在 ROM 集成电路中的信息是永久的或是非易失的(nonvolatile)。即当关掉该电路的电源之后,所存储的信息并不丢失。ROM 有很多类型。目前有三种 ROM 电路在广泛使用,即掩膜 ROM,PROM(一次可编程只读存储器),EPROM(可擦除的可编程只读存储器)。掩膜 ROM 电路是通过半导体制造工艺来实现其数据模式编程的,即所谓的掩膜编程,它的数据是在生产 ROM 芯片时由厂家写入芯片的,一旦芯片被编程,其内容就永远不能被改变。由于这一特点以及进行掩膜编程的造价方面的原因,掩膜 ROM 主要应用于数据将不再改变且具有较大使用量的场合。PROM 和 EPROM 的数据位模式是由用户写入的,编程通常是通过 EPROM 编程器实现。PROM 一旦被编程,其内容就不能再改变。因此它被称为一次可编程只读存储器;EPROM 中的内容可以用紫外线光线照射方法进行擦除,即可以清除已编程的位模式而使该电路恢复到它的未编程状态。通过这种擦除和再编程操作,可以使该电路多次使用。由于它的价格相对较高,EPROM 经常用于产品的早期设计期间。

1. 掩膜 ROM

掩膜 ROM 也称为固定只读存储器,采用掩模工艺制成,因此,其内容由厂方生产时写入,用户只能读出使用而不能改写。只读存储器实质上是一种单向导通的开关阵列,可由二极管构成,也可由 MOS 管或双极型晶体管构成。

(1) 由二极管构成的只读存储器

由二极管构成的只读存储器如图 7-10(a)所示,共有 4 个存储单元,每个单元 4 位,

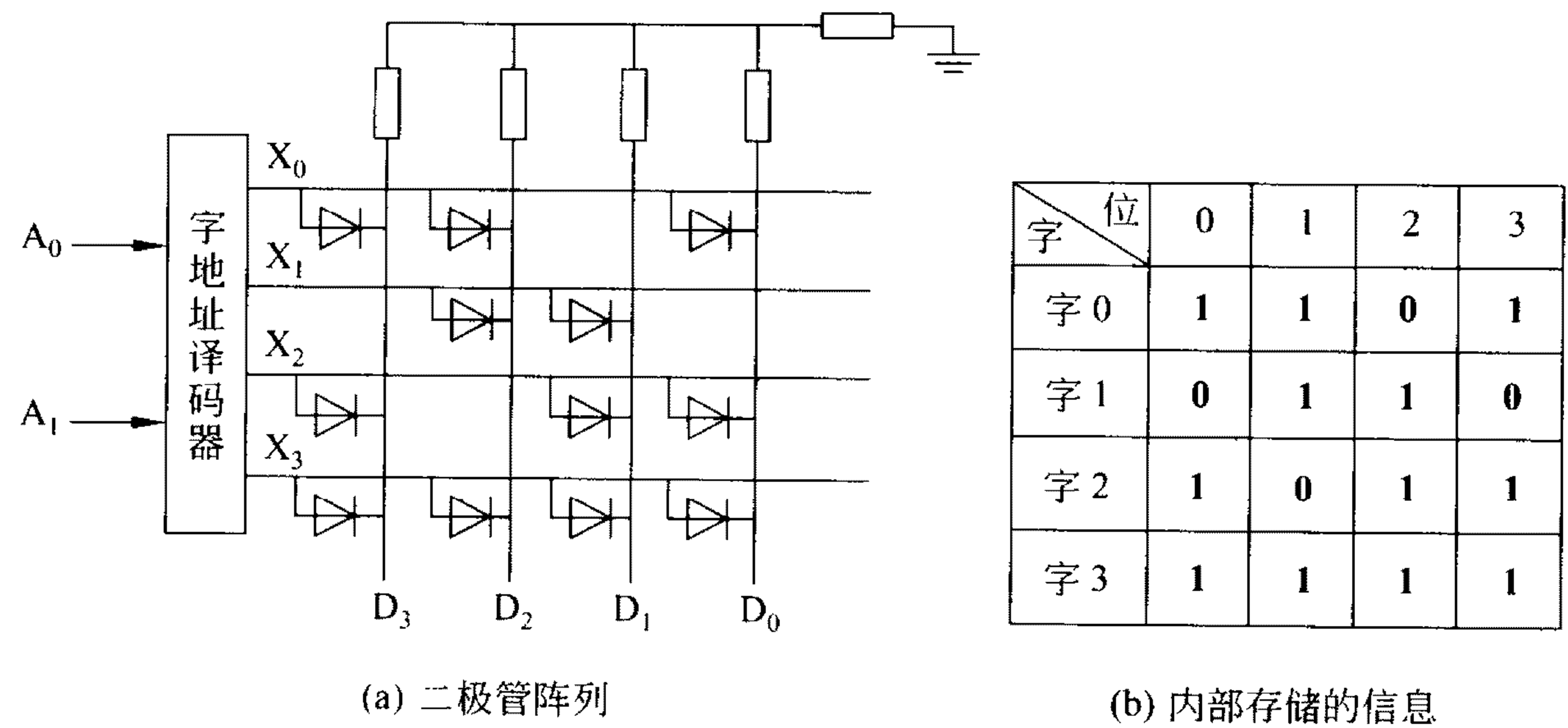


图 7-10 由二极管构成的字选方式只读存储器

采用字译码方式。其中有些位有二极管,有些位没有二极管。这样,若有二极管的位存1,则没有二极管的位存0,其内容如图7-10(b)所示。设给定地址 $A_1 A_0 = 10$,表示选中字2,即 X_2 输出高电平,这样有二极管的位输出高电平,没有二极管的位输出低电平,即读出 $D_3 D_2 D_1 D_0 = 1011$ 。

(2) 复合译码方式的 MOS 管只读存储器

复合译码方式的 MOS 管只读存储器如图7-11所示,用 MOS 管构成导电通路。10位地址分成行、列两部分,分别送 X 地址译码器和 Y 地址译码器。经译码,交叉处为选中单元。若有 MOS 管的位表示存1,则没有 MOS 管的位表示存0。图7-11所示是一种 $1K \times 1\text{bit}$ 只读存储器。

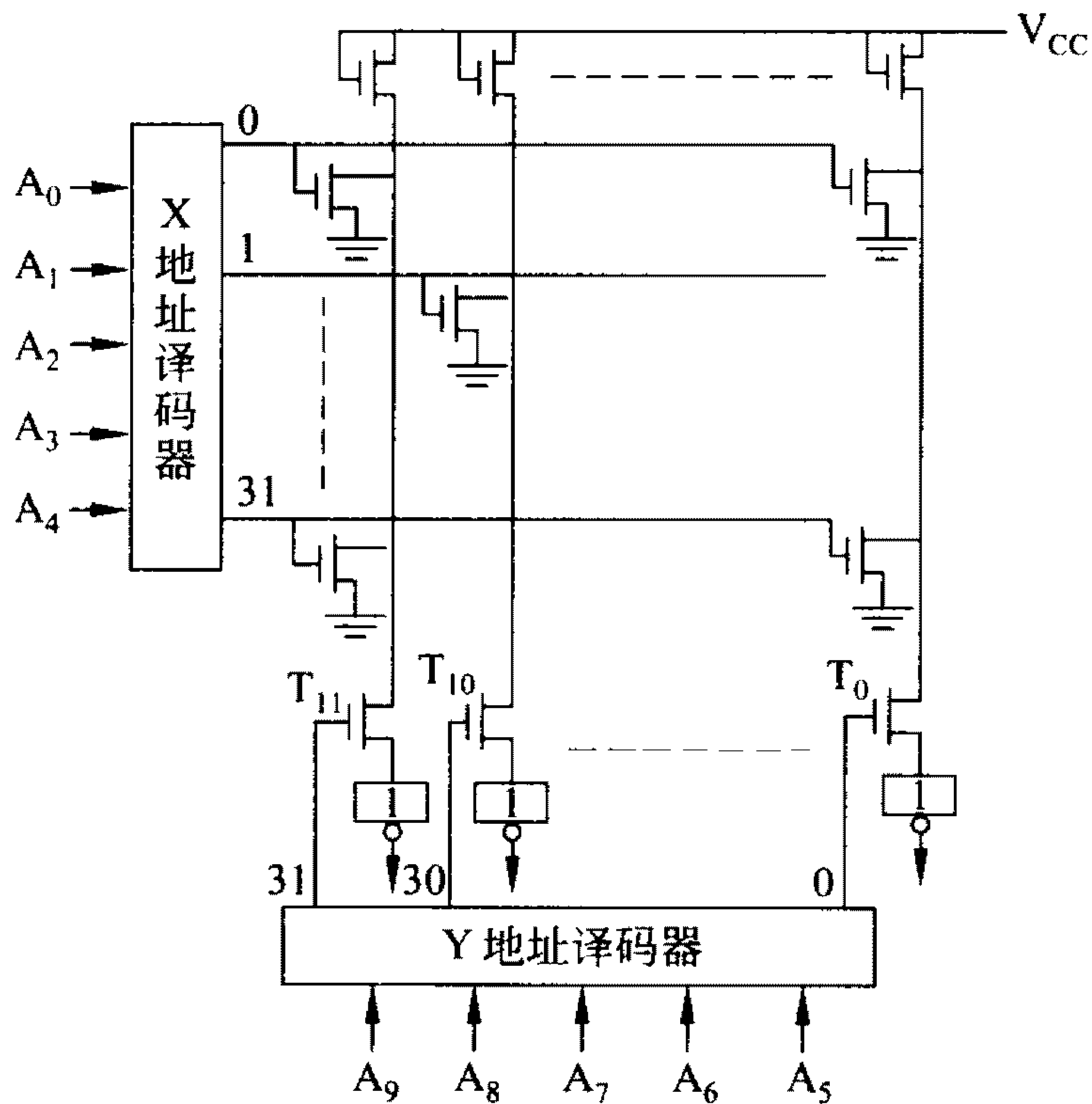


图 7-11 由 MOS 管构成的复合译码方式只读存储器

2. 可写入只读存储器 PROM

可写入只读存储器也称为可编程只读存储器。这种存储器买回时为全0或全1状态,用户可根据自己的需要进行一次性写入编程。

PROM 是将熔断丝或熔合丝串联在 ROM 单元电路中。编程写入时,根据需要(写入1或者0)使其通过一个大的电流,让熔断丝熔断开路或者熔合丝熔合短路。编程写入由专门的电路进行。一旦写入,只能读出使用,不能再修改。

3. 可擦除的可编程只读存储器 EPROM

可擦除的可编程只读存储器是指其中的内容可以通过特殊手段擦去,然后重新写入。其基本单元电路常采用浮空栅雪崩注入式 MOS 电路,简称为 FAMOS。它与 MOS 电路相似,是在 N 型基片上生长出两个高浓度的 P 型区,通过欧姆接触分别引出源极 S 和漏极 D。在源极和漏极之间有一个多晶硅栅极浮空在 SiO_2 绝缘层中,与四周无直接电气连接。这种电路以浮空栅极是否带电来表示存1或者0。浮空栅极带电后(譬如负电

荷),就在其下面,源极和漏极之间感应出正的导电沟道,使 MOS 管导通,即表示存入 1 或者 0。若浮空栅极不带电,则不形成导电沟道,MOS 管不导通,即存入 0 或者 1。

一般在源极和漏极之间加一脉冲电压(24V),可使源漏极之间瞬时击穿,发生雪崩效应。雪崩效应期间,能量大的电子进入浮空栅,雪崩效应结束,电子被困入浮空栅。

改写时可用紫外线照射,使浮空栅中的电子获得足够的能量而散失。然后重新写入。

4. 电擦除只读存储器 E²PROM

电擦除只读存储器通过一定的电压(或电流)来擦除其中的信息,然后重新写入。它的主要特点是能在应用系统中在线改写,断电后信息保存,因此目前得到广泛的应用。

E²PROM 基本存储单元电路的工作原理与 EPROM 相似。它是在 EPROM 基本单元电路的浮空栅的上面再生成一个浮空栅。前者称为第一级浮空栅,后者称为第二级浮空栅。可给第二级浮空栅引出一个电极,使第二级浮空栅栅极接某一电压 V_c。若 V_c 为正电压,第一浮空栅栅极与漏极之间产生隧道效应,使电子注入第一极浮空栅,即编程写入。若使 V_c 为负电压,强使第一级浮空栅栅极的电子散失,即擦除。擦除后可重新写入。

E²PROM 的编程与擦除电流很小,可用普通电源供电,而且擦除可按字节进行。字节编程和擦除时间大约为几个毫秒。

5. 只读存储器举例

目前,只读存储器的种类很多,常用的有 2764、27128、27256、27512、2864、28128、28256 等。

27128 的内部组成与特点:

27128 是一种 16K×8bit 的紫外线擦除可改写只读存储器,采用 HMOS 工艺制成,速度快,读取时间约 200 ns。它有 28 个引脚,采用双列直插式结构,其引脚分布与内部组成如图 7-12 所示。数据线 8 位,地址线 14 位。最大工作电流为 100mA,最大静止等待电流为 40mA,编程电压为 21V,编程负脉冲宽度为 50ms。正常工作电压由单一+5V 电源供电。

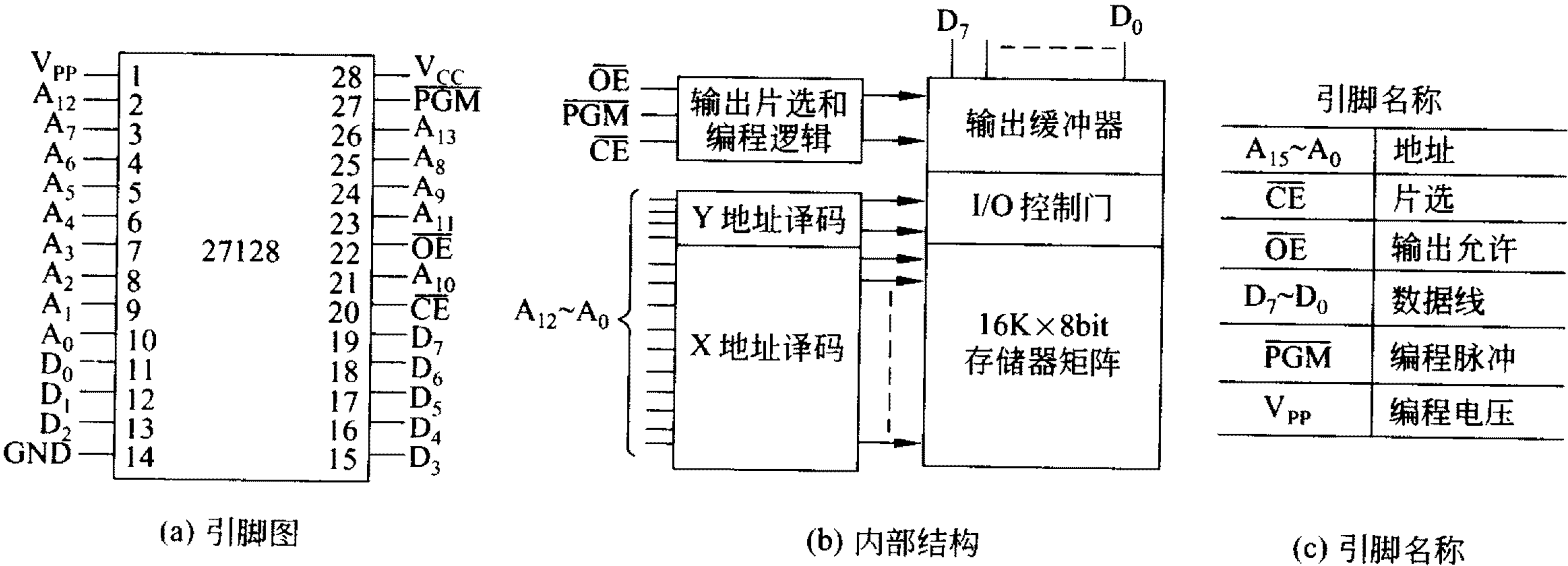


图 7-12 27128 内部框图

7.3 微型计算机系统存储器组织

7.3.1 存储器的扩展技术

由于 80386/80486 微处理器要保持与 8086 等微处理器兼容,这就要求在进行存储器系统设计时必须满足单字节、双字节和 4 字节(即 8 位、16 位或 32 位)等不同数据位的访问。当单个存储芯片的容量不能满足系统要求时,需多片组合起来以扩展字长(位扩展)或字数(字扩展)。

1. 存储器容量的扩展

由于存储芯片的容量是有限的,一个存储体往往是要由一定数量的芯片构成的,我们首先必须知道所用存储芯片的总数量。根据存储器所要求的容量和我们选定的存储芯片的容量,就可以计算出总的芯片数。即:

$$\text{总片数} = \frac{\text{总容量}}{\text{容量/片}}$$

例如:存储器容量为 $8\text{K} \times 8\text{bit}$,若选用 2114 芯片($1\text{K} \times 4\text{bit}$),则需要:

$$\frac{8\text{K} \times 8\text{bit}}{1\text{K} \times 4\text{bit}} = 8 \times 2 = 16 \text{ 片}$$

(1) 位扩展

位扩展指只在位数方向扩展(加大字长),而芯片的字数和存储器的字数是一致的。位扩展的连接方式是将各存储芯片的地址线、片选线和读/写线相应地并联起来,而将各芯片的数据线单独列出。

例如用 $64\text{K} \times 1\text{bit}$ 的 SRAM 芯片组成 $64\text{K} \times 8\text{bit}$ 的存储器,所需芯片数为:

$$\frac{64\text{K} \times 8\text{bit}}{64\text{K} \times 1\text{bit}} = 1 \times 8 = 8 \text{ 片}$$

在这种情况下,CPU 将提供 16 根地址线($2^{16} = 65536$),8 根数据线与存储器相连;而存储芯片仅有 16 根地址线,1 根数据线。具体的连接方法是:8 个芯片的地址线 $A_{15} \sim A_0$ 分别连在一起,各芯片的片选信号 $\overline{\text{CS}}$ 以及读/写控制信号线也都分别连到一起,只有数据线 $D_7 \sim D_0$ 各自独立,每片代表一位,如图 7-13 所示。

当 CPU 访问该存储器时,其发出的地址和控制信号同时传给 8 个芯片,选中每个芯片的同一单元,其单元的内容被同时读至数据总线的相应位,或将数据总线上的内容分别同时写入相应单元。

(2) 字扩展

字扩展是指仅在字数方向扩展,而位数不变。字扩展将芯片的地址线、数据线、读/写线并联,由片选信号来区分各个芯片。

如用 $16\text{K} \times 8\text{bit}$ 的 SRAM 组成

$64\text{K} \times 8\text{bit}$ 的存储器,所需芯片数为:

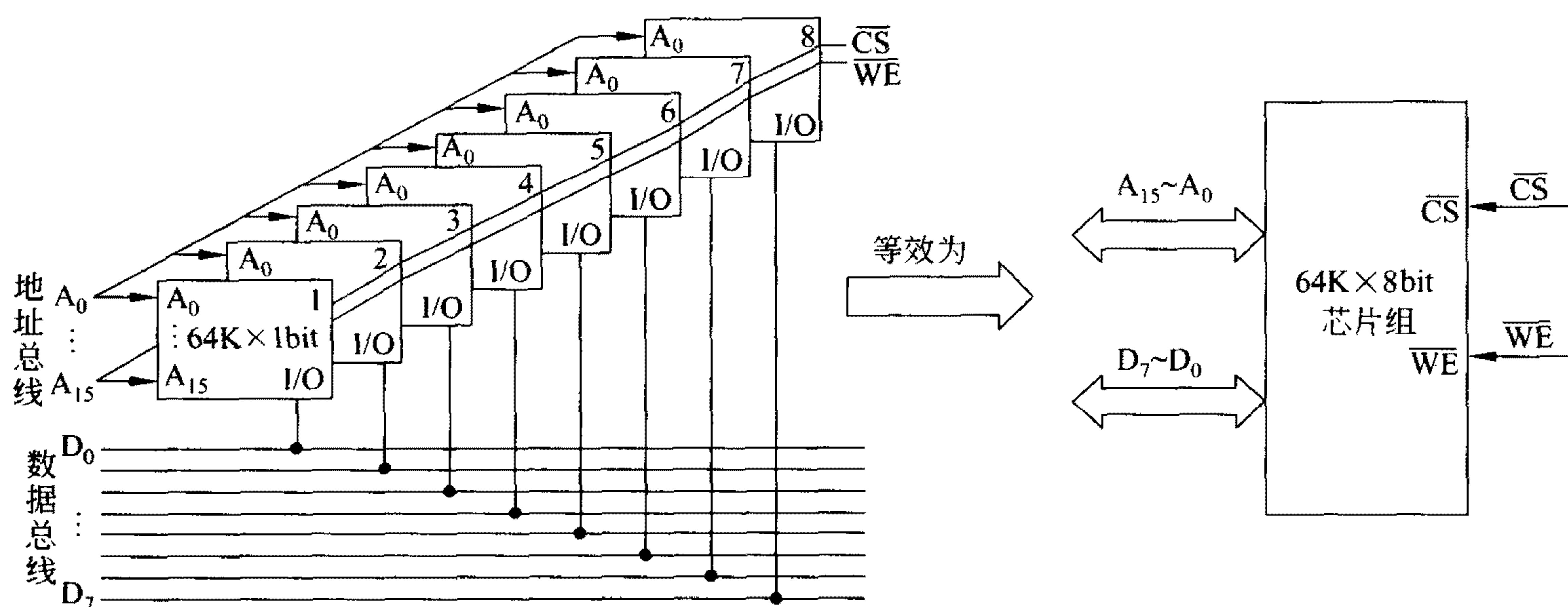


图 7-13 位扩展连接举例

$$\frac{64\text{K} \times 8\text{bit}}{16\text{K} \times 8\text{bit}} = 4 \times 1 = 4 \text{ 片}$$

在这种情况下,CPU 将提供 16 根地址线,8 根数据线与存储器相连;而存储芯片仅有 14 根地址线,8 根数据线。四个芯片的地址线 $A_{13} \sim A_0$,数据线 $D_7 \sim D_0$ 及读/写控制信号 $\overline{\text{WE}}$ 即都是同名信号并联在一起,高位地址线 A_{14} 、 A_{15} 经过一个地址译码器产生四个片选信号 $\overline{\text{CS}}_i$,分别选中四个芯片中的一个,如图 7-14 所示。

在同一时间内四个芯片中只能有一个芯片被选中。四个芯片的地址分配如下:

第 1 片	最低地址	0000H
	最高地址	3FFFH
第 2 片	最低地址	4000H
	最高地址	7FFFH
第 3 片	最低地址	8000H
	最高地址	BFFFH
第 4 片	最低地址	C000H
	最高地址	FFFFH

(3) 字和位同时扩展

当构成一个容量较大的存储器时,往往需要在字数方向和位数方向上同时扩展,这是将前两种扩展组合起来,实现起来也是很容易。

图 7-15 表示用 8 片 $16\text{K} \times 4\text{bit}$ 的 SRAM 芯片组成 $64\text{K} \times 8\text{bit}$ 存储器的示意图。

不同的扩展方法可以得到不同容量的存储器。在选择存储芯片时,一般应尽可能使用集成度高的存储芯片来满足总的存储容量的要求,这样可以减少成本,还可以减轻系统负担,缩小存储器模块的尺寸。

2. 存储芯片的地址分配和片选

CPU 与存储器连接时,特别是在扩展存储容量的场合下,主存的地址分配就是一个重要的问题;确定地址分配后,又有一个产生选择存储芯片的片选信号的问题。

CPU 要实现对存储单元的访问,首先要选择存储芯片,即进行片选;然后再从选中的

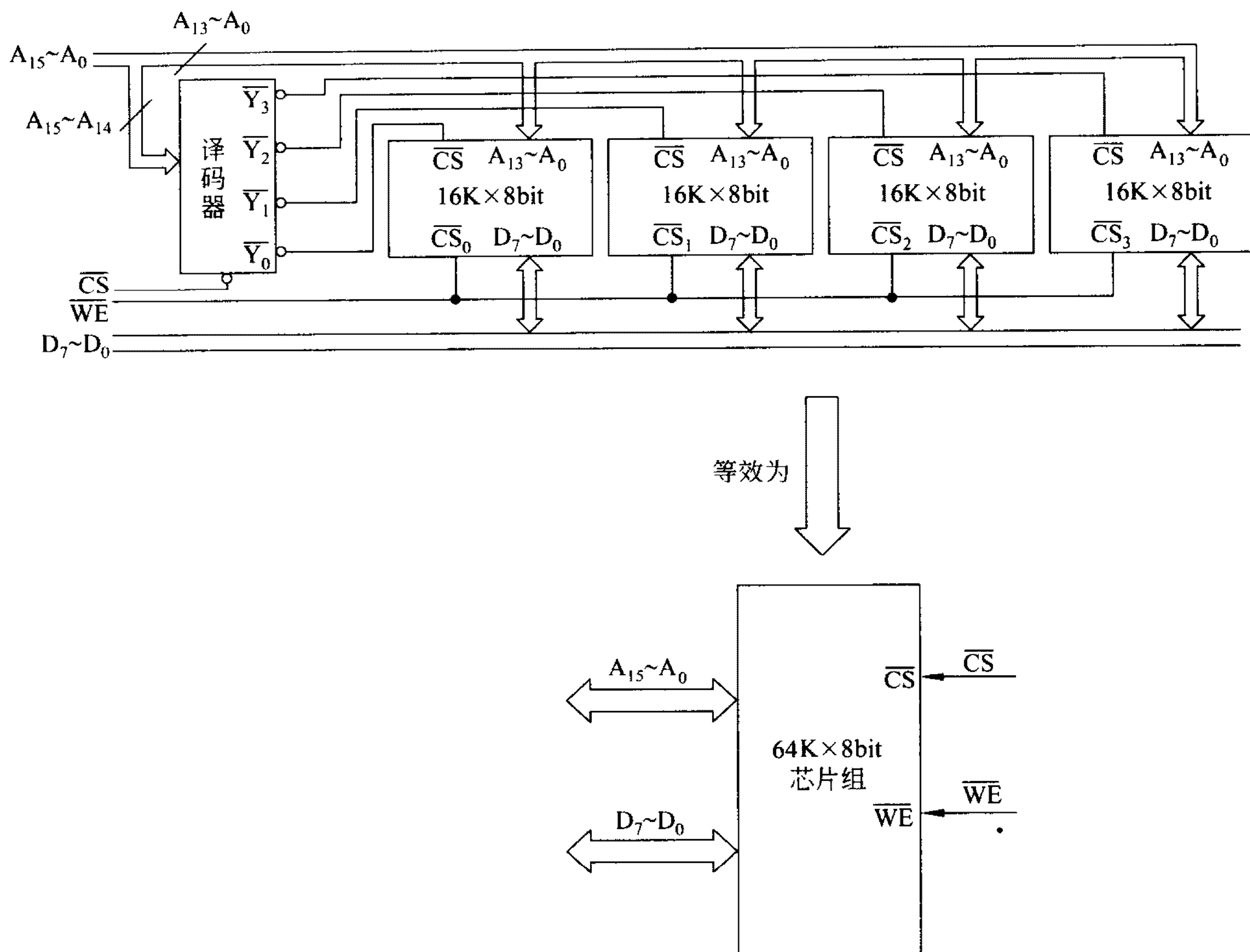


图 7-14 字扩展连接举例

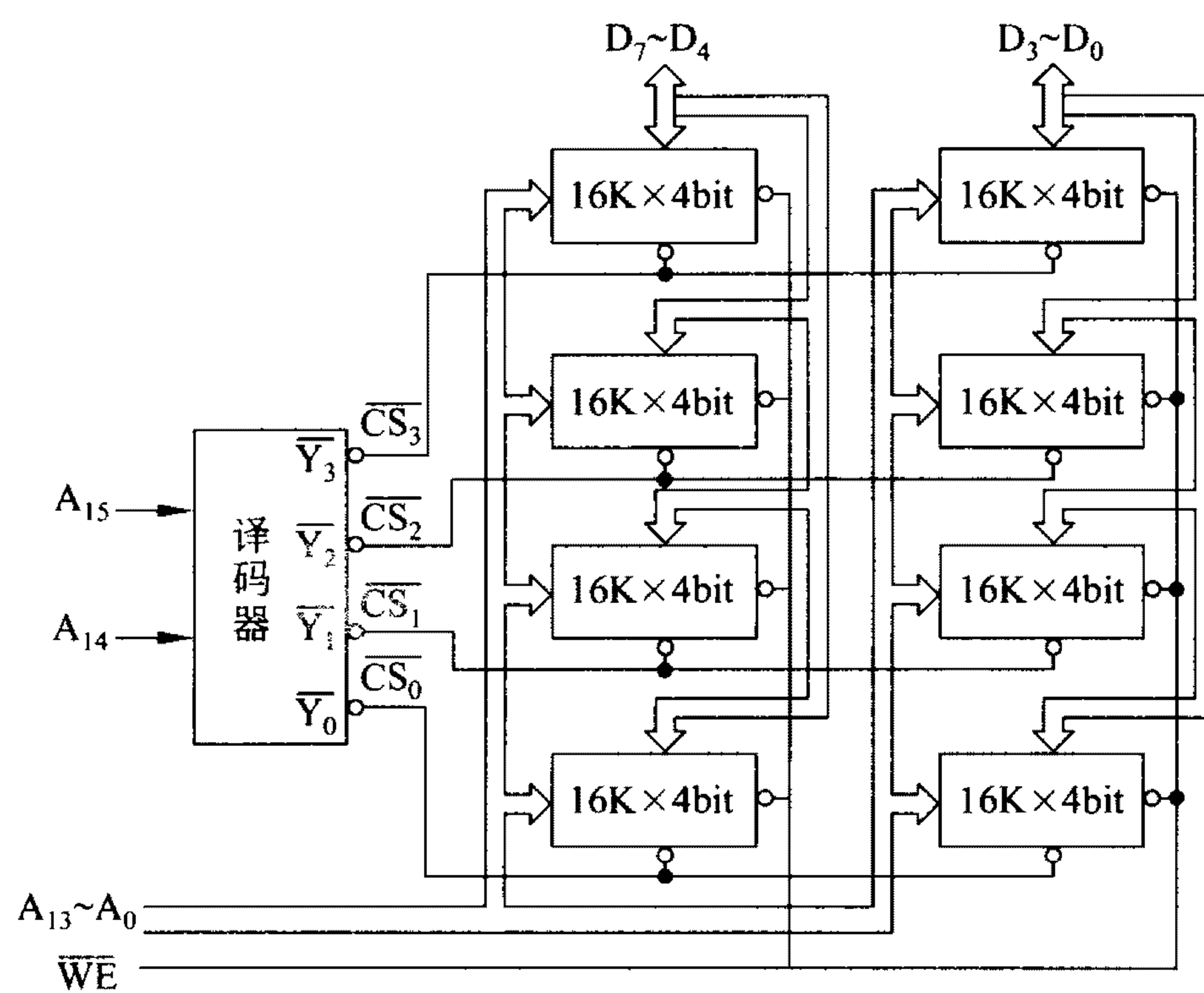


图 7-15 字和位同时扩展连接举例

芯片中依地址码选择出相应的存储单元,以进行数据的存取,这称为字选。片内的字选是由 CPU 送出的 N 条低位地址线完成的,地址线直接接到所有存储芯片的地址输入端 (N 由片内存储容量 2^N 决定),而片选信号则是通过高位地址得到的。实现片选的方法可分为三种:即线选法、全译码法和部分译码法。

(1) 线选法

线选法就是用除了片内寻址外的高位地址线直接(或经反相器)接至各个存储芯片的片选端,当某地址线信息为“0”时,就选中与之对应的存储芯片。请注意,这些片选地址线每次寻址时只能有一位有效,不允许同时有多位有效,这样才能保证每次只选中一个芯片(或组)。图 7-16 为 4 片 $2\text{K} \times 8\text{bit}$ 芯片用线选法构成的 $8\text{K} \times 8\text{bit}$ 存储器的连接图。各芯片的地址范围如表 7-3 所示,设地址总线有 20 位($A_{19} \sim A_0$)。

表 7-3 线选法的地址分配

芯片	$A_{19} \sim A_{15}$	$A_{14} \sim A_{11}$	$A_{10} \sim A_0$	地址范围(空间)
0 [#]	0...0	1110	00...0 11...1	07000H~077FFH
1 [#]	0...0	1101	00...0 11...1	06800H~06FFFH
2 [#]	0...0	1011	00...0 11...1	05800H~05FFFH
3 [#]	0...0	0111	00...0 11...1	03800H~03FFFH

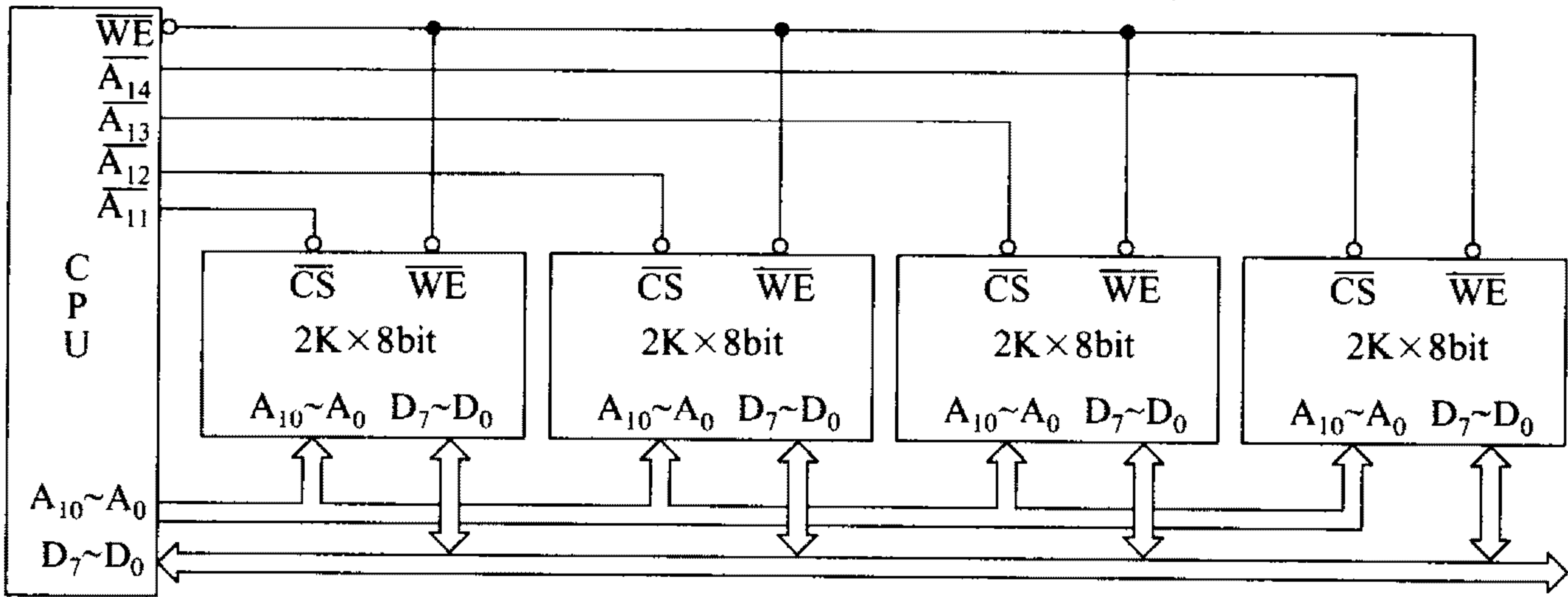


图 7-16 线选法构成的 $8\text{K} \times 8\text{bit}$ 存储器的连接图

线选法的优点是不需要地址译码器,线路简单,选择芯片不需外加逻辑电路,但仅适用于连接存储芯片较少的场合。同时,线选法不能充分利用系统的存储器空间,且把地址空间分成了相互隔离的区域,给编程带来了一定的困难。

(2) 全译码法

全译码法将片内寻址外的全部高位地址线作为地址译码器的输入,把经译码器译码后的输出作为各芯片的片选信号,将它们分别接到存储芯片的片选端,以实现对各芯片的选择。如前述 4 片 $2\text{K} \times 8\text{bit}$ 的存储芯片用全译码法构成 $8\text{K} \times 8\text{bit}$ 存储器,各芯片

的地址范围如表 7-4 所示。

表 7-4 全译码法的地址分配

芯 片	$A_{19} \sim A_{13}$	$A_{12} \sim A_{11}$	$A_{10} \sim A_0$	地址范围(空间)
$0^{\#}$	$0 \cdots 0$	00	$00 \cdots 0$ $11 \cdots 1$	00000H ~ 007FFH
$1^{\#}$	$0 \cdots 0$	01	$00 \cdots 0$ $11 \cdots 1$	00800H ~ 00FFFH
$2^{\#}$	$0 \cdots 0$	10	$00 \cdots 0$ $11 \cdots 1$	01000H ~ 017FFH
$3^{\#}$	$0 \cdots 0$	11	$00 \cdots 0$ $11 \cdots 1$	01800H ~ 01FFFH

全译码法的优点是每片(或组)芯片的地址范围是唯一确定的,而且是连续的,也便于扩展,不会产生地址重叠的存储区,但全译码法对译码电路的要求较高,如上例中, $A_{11} \sim A_{19}$ 共 9 根地址线都要参与译码。

(3) 部分译码

在系统中如果不要求提供 CPU 可直接寻址的全部存储单元,则可采用线选法和全译码法相结合的方法,这就是部分译码法。所谓部分译码即:用除了片内寻址外的高位地址的一部分来译码产生片选信号。用 4 片 $2K \times 8\text{bit}$ 的存储芯片组成 $8K \times 8\text{bit}$ 存储器,需要四个片选信号,因此只要用两位地址线来译码产生。

由于寻址 $8K \times 8\text{bit}$ 存储器时未用到高位地址 $A_{19} \sim A_{13}$,所以只要 $A_{12} = A_{11} = 0$,而无论 $A_{19} \sim A_{13}$ 取何值,均选中第一片,只要 $A_{12} = 0, A_{11} = 1$,而无论 $A_{19} \sim A_{13}$ 取何值,均选中第二片,……也就是说,8KBRAM 中的任一个存储单元,都对应有 $2^{(20-13)} = 2^7$ 个地址,这种一个存储单元出现多个地址的现象称地址重叠。

从地址分布来看,这 8KB 存储器实际上占用了 CPU 全部的空间(1MB)。每片 $2K \times 8\text{bit}$ 的存储芯片有 $1\text{MB}/4 = 256\text{KB}$ 的地址重叠区,见图 7-17 所示。令未用到的高位地址全为 0,这样确定的存储器地址称为基本地址。

本例中 $8K \times 8\text{bit}$ 存储器的基本地址即 00000H~007FFH。部分译码法较全译码法简单,但存在地址重叠区。在实际应用中,存储芯片的片选信号可根据需要选择上述某种方法或几种方法并用。

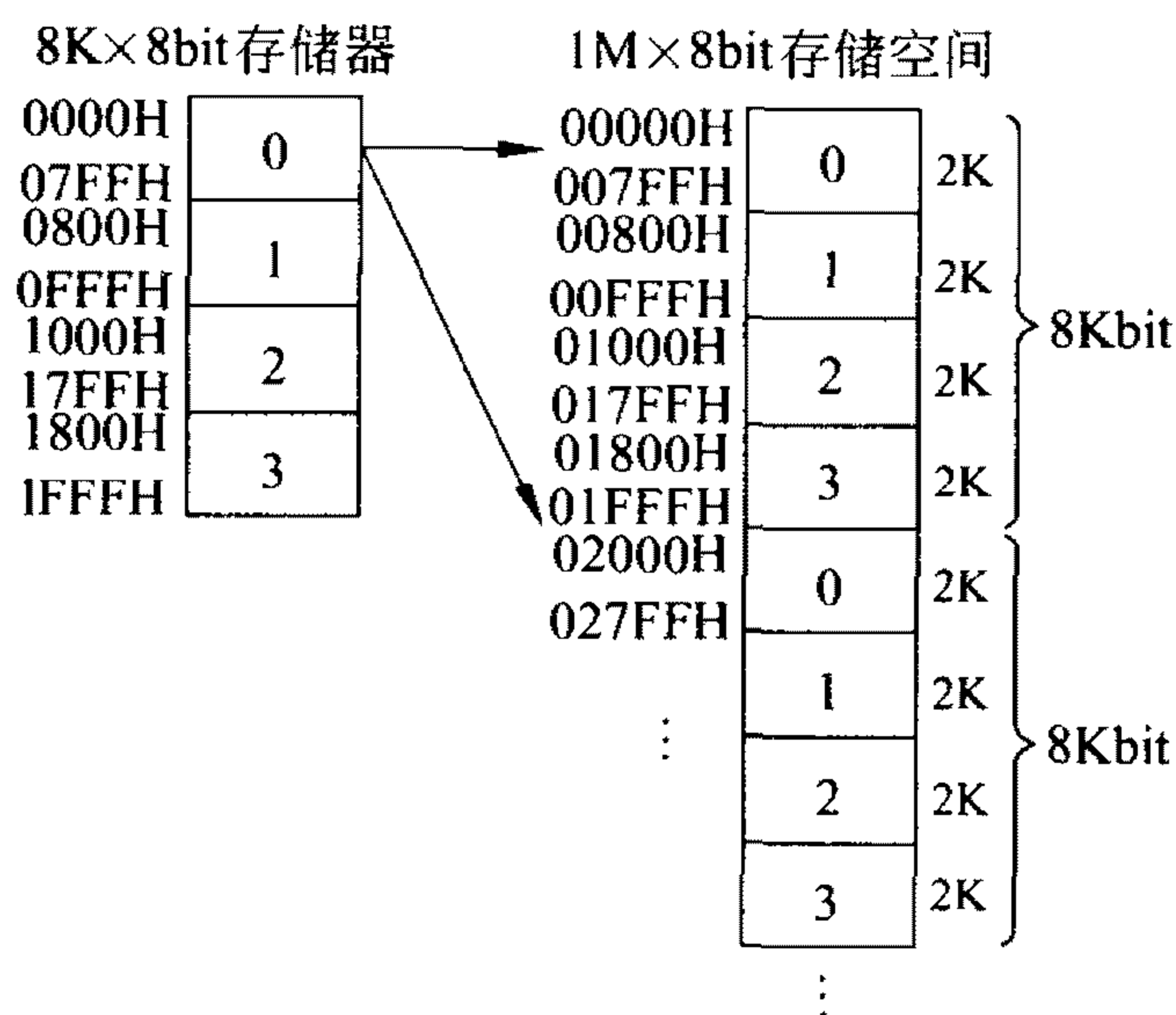


图 7-17 地址重叠区示意图

7.3.2 CPU 与主存储器的连接

在讨论了主存储器的结构之后,进一步了解主存和 CPU 之间的连接是十分必要的。

1. 主存和 CPU 之间的硬连接

主存与 CPU 的硬连接有三组连线：地址总线 (AB)、数据总线 (DB) 和控制总线 (CB)，如图 7-18 所示。此时，我们把主存看作一个黑盒子，存储器地址寄存器 (MAR) 和存储器数据寄存器 (MDR) 是主存和 CPU 之间的接口。MAR 可以接受来自程序计数器的指令地址或来自运算器的操作数地址，以确定要访问的单元。MDR 是向主存写入数据或从主存读出数据的缓冲部件。MAR 和 MDR 从功能上看属于主存，但在小、微型机中常放在 CPU 内。

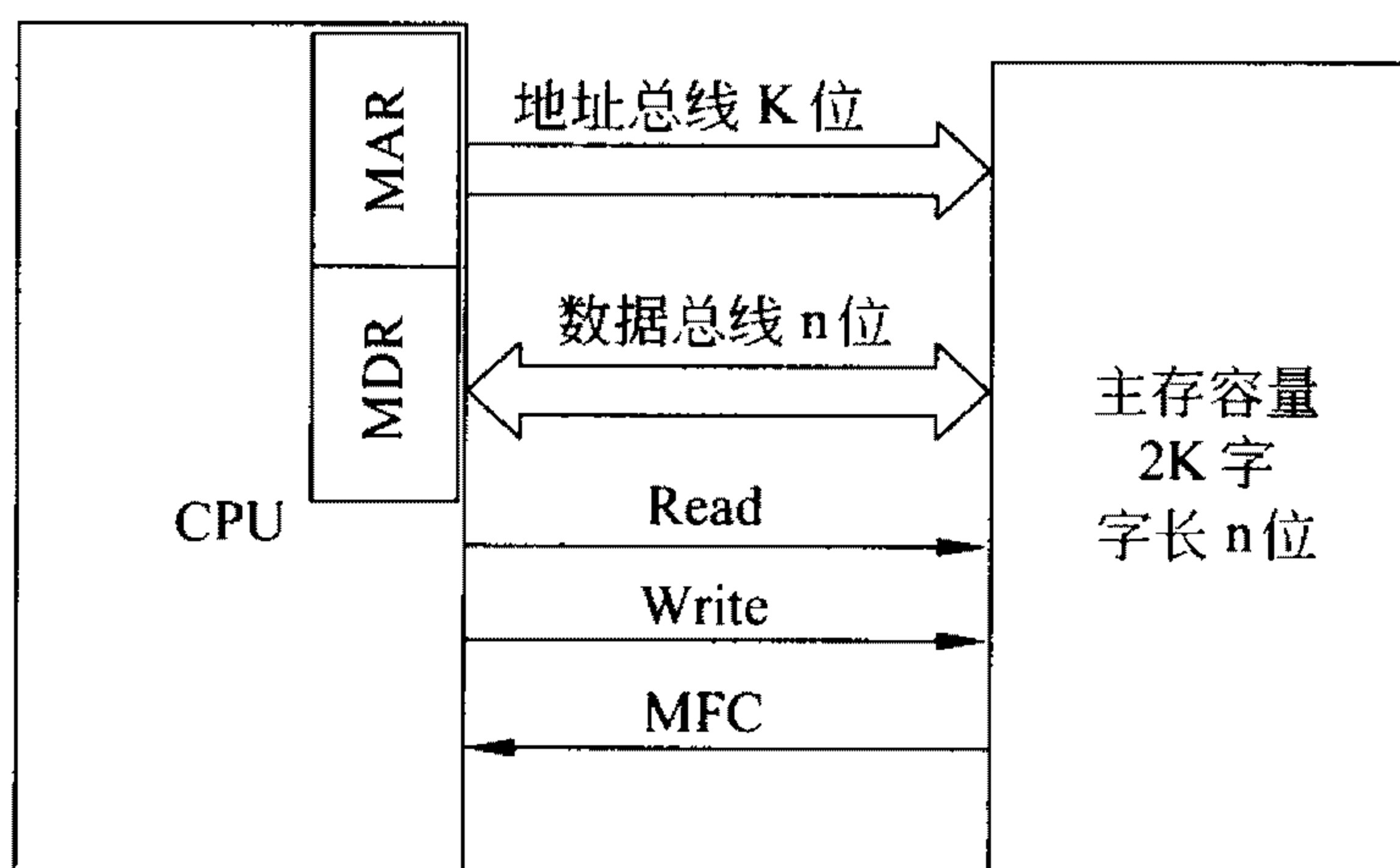


图 7-18 主存和 CPU 的连接

2. CPU 总线的负载能力

在计算机系统中，由于 CPU 往往通过总线与数片存储芯片相连接，而且这些芯片可能是 TTL 器件，也可能是 MOS 器件，所以当构成系统时，能否支持其负载是必须考虑的问题。当 CPU 总线上挂接的器件均为 MOS 器件且数量不多时，一般不会超载。当总线上挂接的器件过多可能使 CPU 超载时，就该考虑总线的驱动问题了。增加 CPU 总线带负载能力的主要方法是在总线上增加缓冲器和驱动器。

用不同的存储芯片组成存储器对总线负载情况的影响是不同的。容量大的存储器芯片对总线的负载小。当总线上的芯片接得很多时系统不但需要接总线驱动器，且负载电容也可能变得很大。

3. CPU 对主存的基本操作

前面所说的 CPU 与主存的硬连接是两个部件之间联系的物理基础，而两个部件之间还有软连接，即 CPU 向主存发出的读或写命令，这才是两个部件之间有效工作的关键。

CPU 对主存进行读/写操作时，首先 CPU 在地址总线上给出地址信号，然后发出相应的读或写命令，并在数据总线上交换信息。读/写的基本操作如下：

① 读。读操作是指从 CPU 送来的地址所指定的存储单元中取出信息，再送给 CPU。其操作过程是：

地址 \rightarrow MAR \rightarrow AB	CPU 将地址信号送至地址总线
Read	CPU 发读命令
Wait for MFC	等待存储器工作完成信号
(MAR) \rightarrow DB \rightarrow MDR	读出信息经数据总线送至 CPU

② 写。写操作是指将要写入的信息存入 CPU 所指定的存储单元中。其操作过程是：

地址 \rightarrow MAR \rightarrow AB	CPU 将地址信号送至地址总线
数据 \rightarrow MDR \rightarrow DB	CPU 将要写入的数据送至数据总线
Write	CPU 发出写命令
Wait for MFC	等待存储器工作完成信号

由于 CPU 和主存的速度存在着差距,所以两者之间的速度匹配是很关键的,通常有两种匹配方式:同步存储器读取和异步存储器读取。

上面给出的读/写基本操作是以异步存储器读取来考虑的,CPU 和主存间没有统一的时钟,由存储器工作完成信号(MFC)通知 CPU 存储器工作已完成。对于读操作,若 $MFC=1$,说明信息已经读出;对于写操作,若 $MFC=1$,说明数据已写入相应的存储单元。

对于同步存储器读取,CPU 和主存采用统一时钟,同步工作,因为主存速度较慢,所以 CPU 与之配合必须放慢速度。在这种存储器中,不需要存储器工作完成信号。

4. DRAM 与 CPU 的连接

SRAM 或 ROM 与 CPU 的连接都比较简单,而 DRAM 由于行、列地址复用一组引脚,所以需要多路转换器;在行地址中,又要能接入刷新地址,因此也要有多路转换器。它与 CPU 间的接口电路如图 7-19 所示。

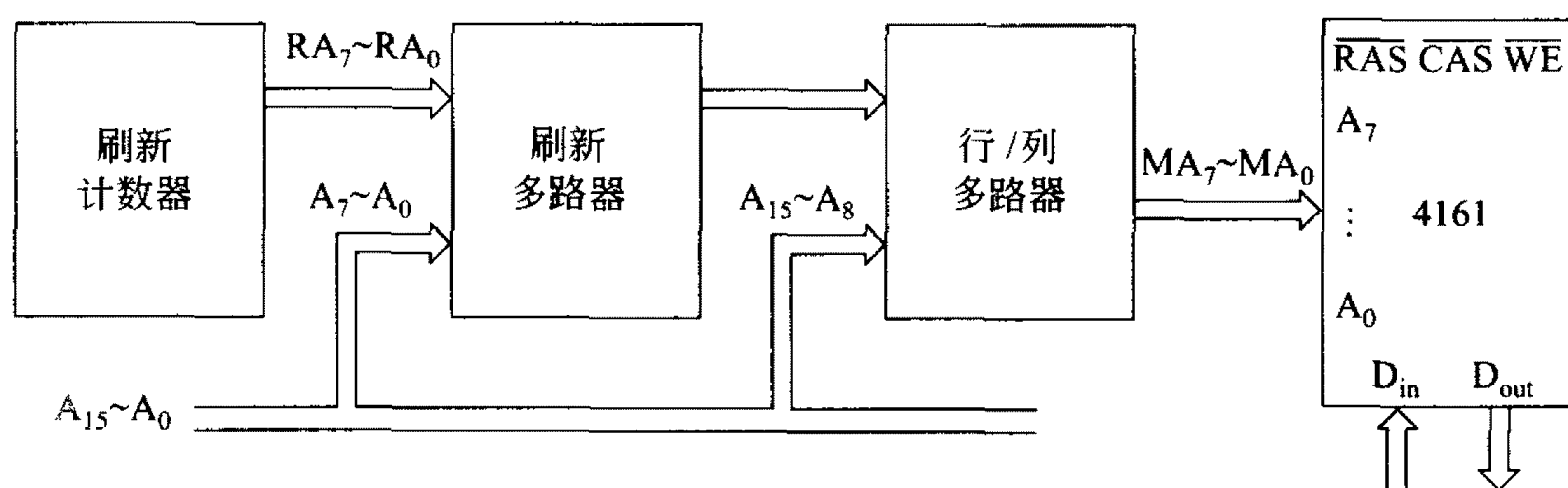


图 7-19 DRAM 和 CPU 间的接口电路

对于容量需要扩展的 DRAM, \overline{RAS} 和 \overline{CAS} 这两个信号还包含有片选的功能。例如:用 4164 组成的 $256K \times 8\text{bit}$ 的存储器,需要 32 片 4164,分成四组,访问它们的行地址选通信号为 $\overline{RAS}_3 \sim \overline{RAS}_0$,列选通信号为 $\overline{CAS}_3 \sim \overline{CAS}_0$,其中 0~3 表示芯片组号。这就需要专门的 \overline{RAS} 和 \overline{CAS} 的生成电路,当存储器在读或写操作时,分别产生 \overline{RAS} 和 \overline{CAS} 信号,控制 DRAM 芯片的工作,而在刷新操作时,同时输出低电平有效的 $\overline{RAS}_0 \sim \overline{RAS}_3$ 信号,对全部 DRAM 芯片执行刷新操作。

有些厂家专门设计了包括刷新支持电路及控制行/列地址复用电路在内的单片集成电路,称为 DRAM 控制器。如 Intel8203 就是其中的一种。8203 的逻辑框图如图 7-20 所示,它是专门用来支持 16KB 或 64KB DRAM 芯片的,提供了地址多路转换、地址选通、刷新控制、刷新/访存仲裁等功能。

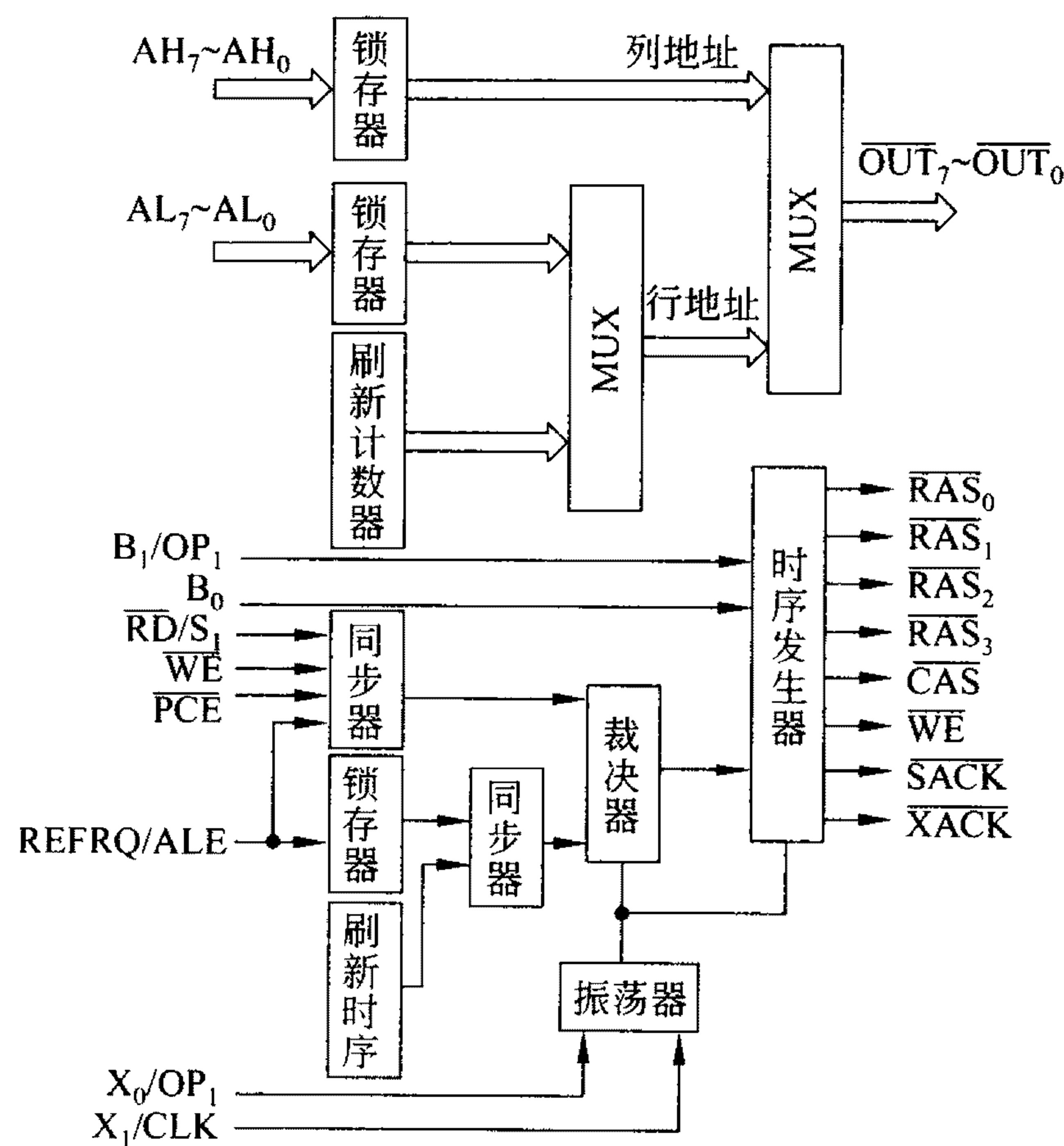


图 7-20 8203DRAM 控制器

8023 内部可以产生刷新时序,但也可由外部输入信号 REFRQ 来产生外部刷新请求,若外部刷新请求间隔小于内部刷新定时,那么刷新完全由外部请求实现。

7.3.3 PC 机的存储器组织

数据总线一次能并行传送的位数称为总线的通路宽度,常见的有 8 位、16 位、32 位、64 位几种。但大多数主存储器常采取字节编址,每次访存允许读/写 8 位,以适应对字符类信息的处理。这就存在着主存与数据总线之间的宽度匹配和存储器接口的问题,下面以 PC 系列微型计算机为例讨论这一问题。

1. 8 位存储器接口

如果数据总线为 8 位(如微型计算机系统 PC 总线),而主存按字节编址,则匹配关系比较简单。对于 8 位(或准 16 位)的微处理器,典型的时序安排是占用 4 个 CPU 时钟周期,称为 T₁~T₄,构成一个总线周期,一个总线周期中读/写 8 位。

8 位的微处理器 8088 提供 RD(读选通)、WR(写选通)和 IO/M(I/O 或存储器控制)等控制信号(最小模式)去控制存储器系统,或者提供 IO/M 与 RD 一起产生的 MRDC(存储器读命令)、IO/M 与 WR 一起产生的 MWTC(存储器写命令)等控制信号(最大模式)

去控制存储器系统。

2. 16 位存储器接口

对于 16 位的微处理器 8086(或 80286),在一个总线周期内可读/写两个字节,即先送出偶地址,然后同时读/写这个偶地址单元和随后的奇地址单元,用低 8 位数据总线传送偶地址单元的数据,用高 8 位数据总线传送奇地址单元的数据,这样读/写的字(16 位)被称为规则字。如果读/写的是非规则字,即从奇地址开始的字,这时需要安排两个总线周期才能实现。

为了实现这样的传送、需要将存储器分为两个存储体,如图 7-21 所示。一个存储体的地址均为偶数,称为偶地址(低字节)存储体,它与低 8 位数据线相连;另一个存储体的地址均为奇数,称为奇地址(高字节)存储体,与高 8 位数据线相连。

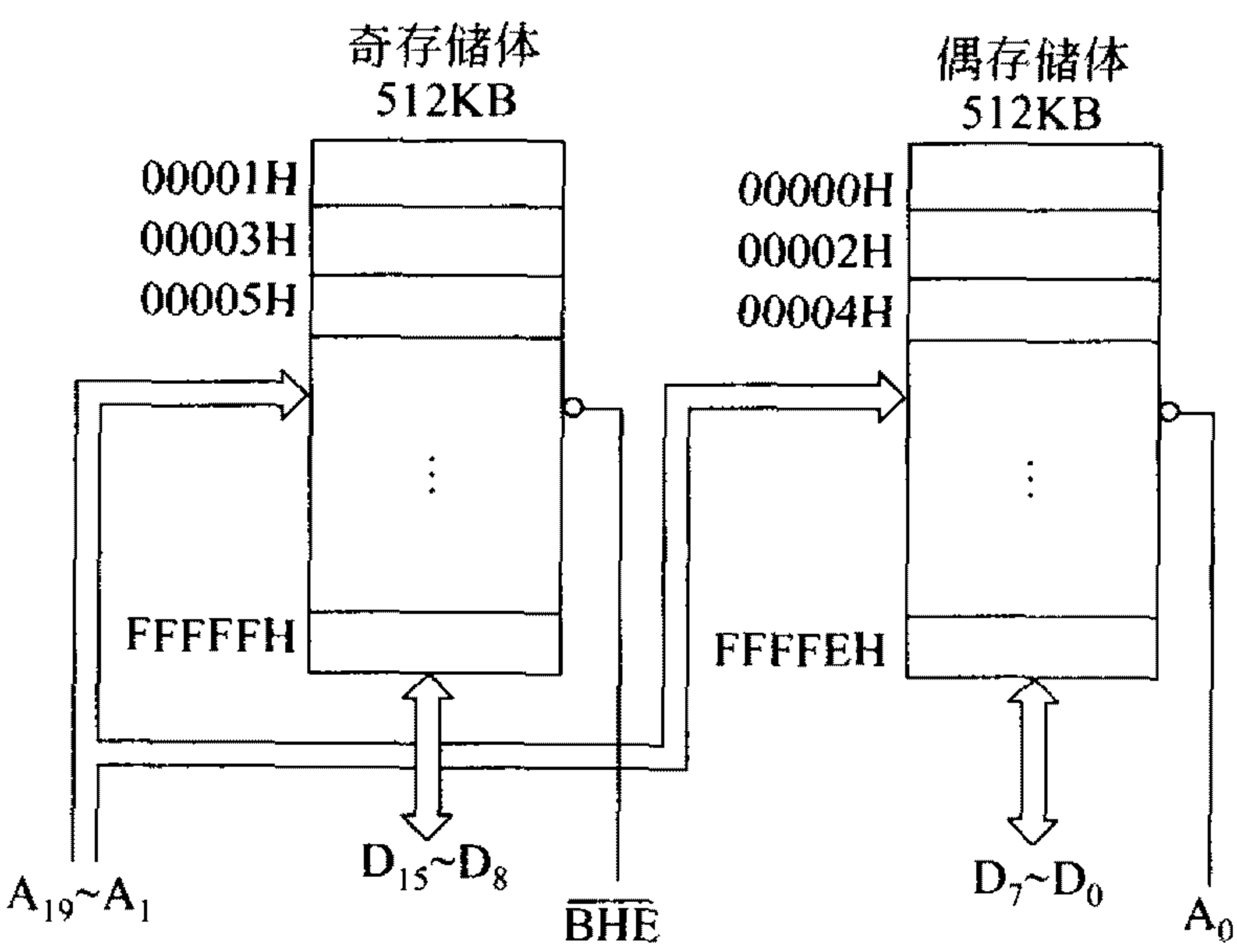


图 7-21 8086 的存储器组织

8086 微处理器的地址线 $A_{19} \sim A_1$ 同时送至两个存储体, \overline{BHE} (高位存储体)和最低位地址线 A_0 用来选择一个或两个存储体进行数据传送。 \overline{BHE} 和 A_0 的选择如表 7-5 所示。

表 7-5 \overline{BHE} 和 A_0 的选择表

\overline{BHE}	A_0	特 征	\overline{BHE}	A_0	特 征
0	0	全字(规则字)传送	0	1	在数据总线高 8 位进行字节传送
1	0	在数据总线低 8 位进行字节传送	1	1	备用

8086 和主存之间可以传送一个字节(8 位)数据,也可以传送一个字(16 位)数据。任何两个连续的字节都可以作为一个字来访问,地址值较低的字节是低位有效字节,地址值较高的字节是高位有效字节。

地址最低位 A_0 决定了字的边界。如果 $A_0 = 0$,则字存放在偶地址边界上,低 8 位有效字节存储于偶地址单元中。例如:

地址	存储单元内容
00500H	24H
00501H	65H

低 8 位有效字节在地址 00500H 单元中,故字 6524H 是存放在偶地址边界上。
若 $A_0=1$,则字存放在奇地址边界上,例如:

地址	存储单元内容
007A1H	39H
007A2H	A7H

字 A739H 是存放在奇地址边界上。

当存取规则字时,地址线送出偶地址($A_0=0$),并同时让 \overline{BHE} 有效。于是同时选中两个存储体,分别读出高字节与低字节,共 16 位,在一个总线周期同时传送。

对于规则字进行读/写,仅需一次访问存储器,而对于非规则字进行读/写,就需要两次访问存储器才能实现,而且每次都应忽略掉不需要的半个字。图 7-22 给出了各种信息的传送方法。其中图 7-22(a)为偶地址字节传送,图 7-22(b)为奇地址字节传送,图 7-22(c)为偶地址字传送,图 7-22(d)和(e)为奇地址字传送,图中有阴影的矩形框,代表本次读写的存储单元。

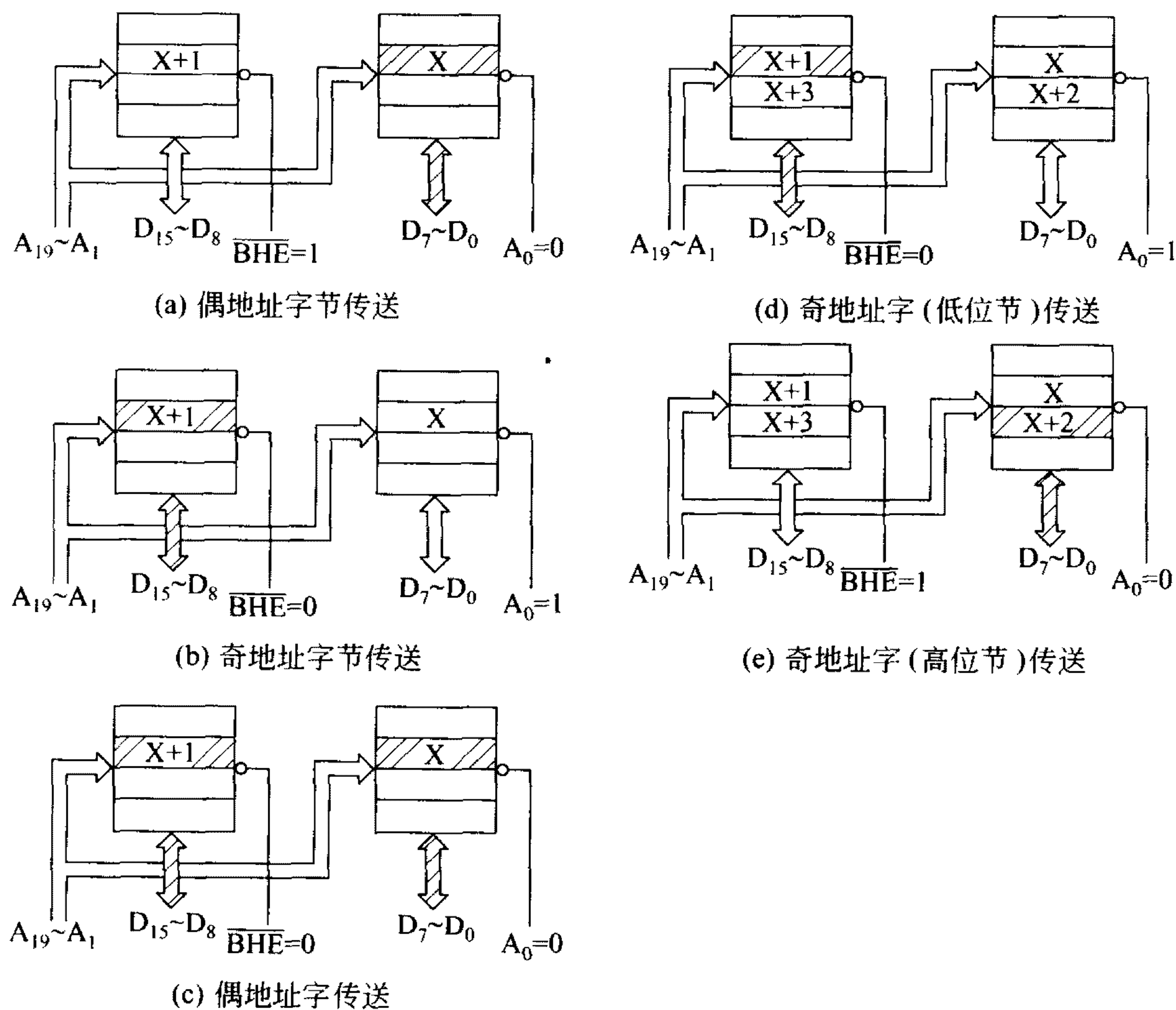


图 7-22 各种信息的传送方法

处理存储体选择的最有效的方式是让每个存储体有独立的写选通信号,但通常不必要产生独立的读选通信号,在一次读操作中总提供 16 位数据给数据总线,微处理器将根据需要忽略它不需要的 8 位部分,这样做不会产生任何冲突或特殊问题。

将 A_0 和 \overline{WR} 组合在一起产生低位存储体选择信号(LWR),将 \overline{BHE} 和 \overline{WR} 组合在一起

产生高位存储体选择信号(HWR)。

3. 32 位存储器接口

32 位微处理器的存储器系统由 4 个存储体组成,每个存储体的存储空间为 1GB,存储体选择通过选择信号 \overline{BE}_0 、 \overline{BE}_1 、 \overline{BE}_2 、 \overline{BE}_3 实现。如果要传送一个 32 位数,那么 4 个存储体都被选中;若要传送一个 16 位数,则有 2 个存储体(通常是 \overline{BE}_3 和 \overline{BE}_2 或 \overline{BE}_1 和 \overline{BE}_0)被选中;若传送的是 8 位数,只有一个存储体被选中。

32 位微处理器的存储器组织如图 7-23 所示,在对 32 位地址进行译码时,最低地址位 A_1 、 A_0 不作考虑,它用来产生存储体允许信号。写选通信号的产生电路如图 7-24 所示。

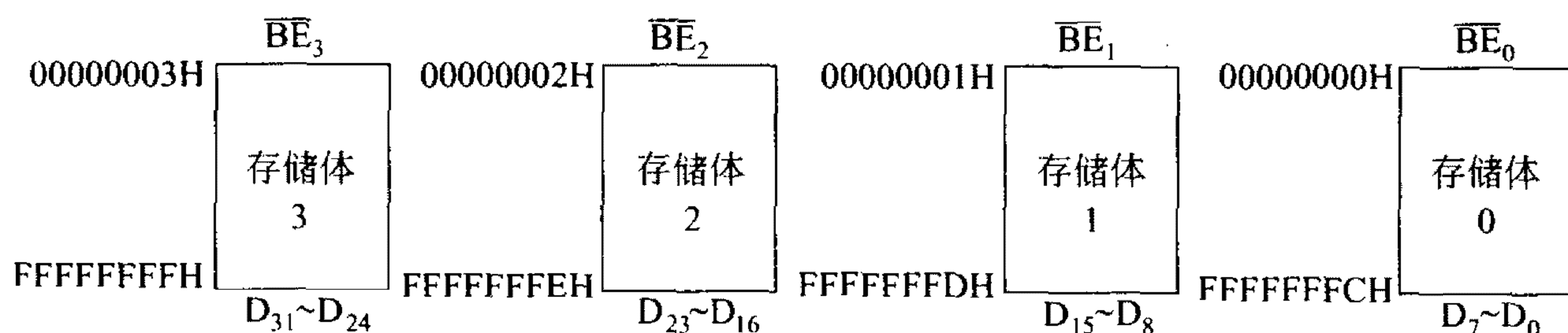


图 7-23 32 位微处理器的存储器组织

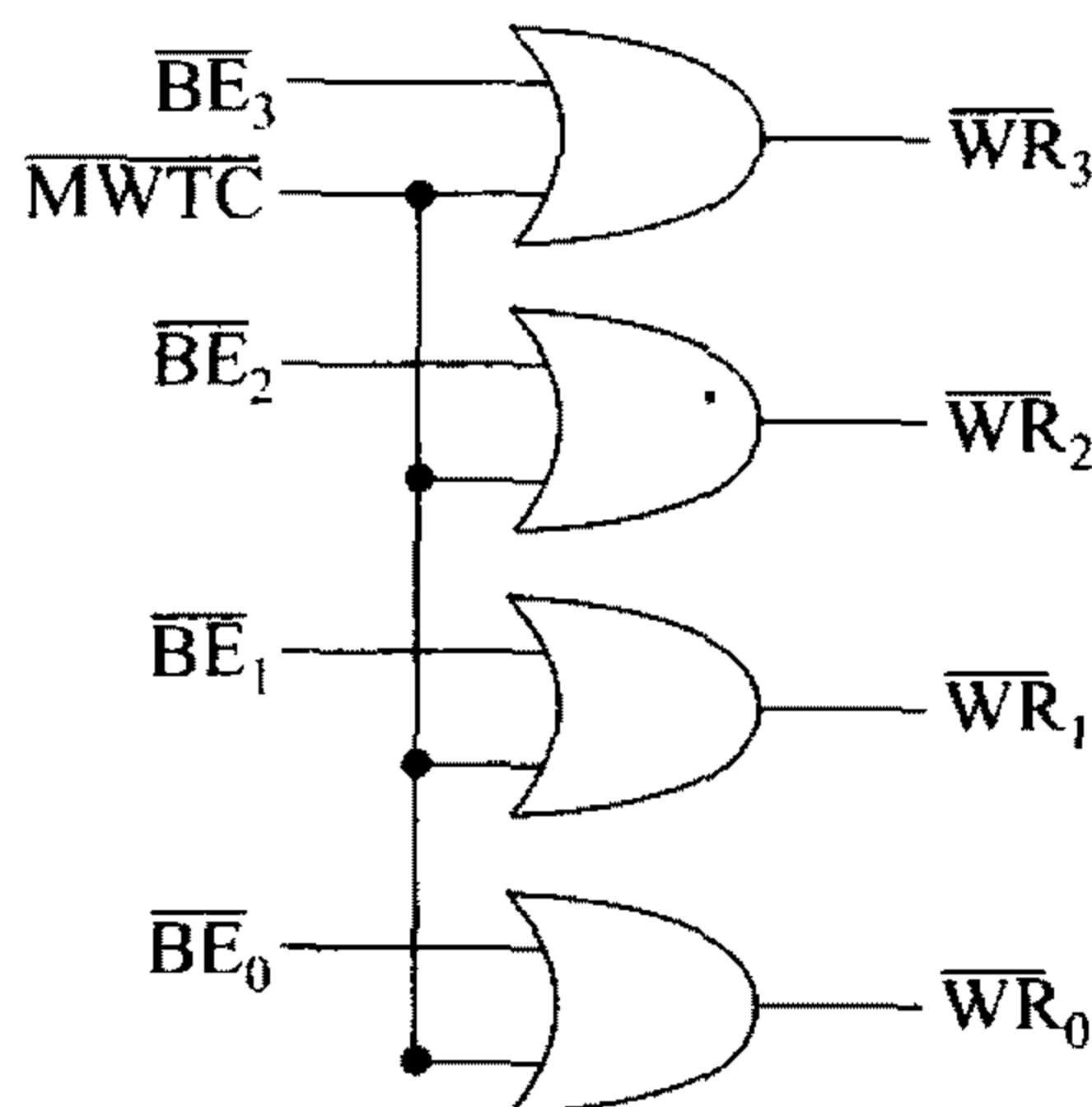


图 7-24 32 位微处理器的写选通信号

4. 64 位存储器接口

64 位微处理器的存储系统由 8 个存储体组成,每个存储体的存储空间为 512MB (Pentium)或 8GB(Pentium Pro),存储体选择通过选择信号 $\overline{BE}_7 \sim \overline{BE}_0$ 实现。如果要传送一个 64 位数,那么 8 个存储体都被选中;如果要传送一个 32 位数,那么 4 个存储体都被选中;若要传送一个 16 位数,则有 2 个存储体被选中;若传送的是 8 位数,只有一个存储体被选中。

64 位微处理器的存储器组织与前述 32 位微处理器相似,在此不再重复。图 7-25 为 64 位微处理器的写选通电路。图 7-26 给出了 Pentium 微处理器的地址总线与 64 位、32 位、16 位和 8 位存储器的接口信号示意图。

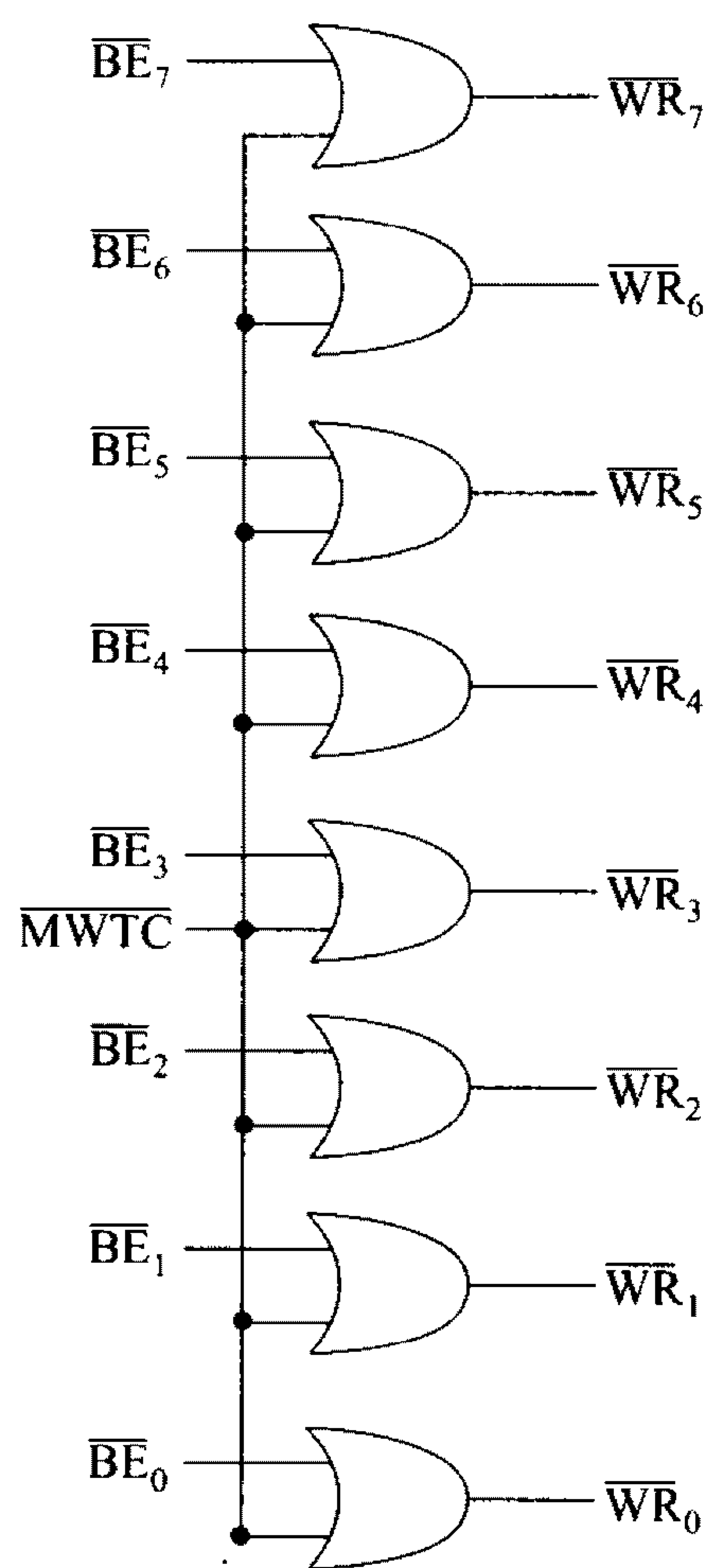


图 7-25 64 位微处理器的写选通信号

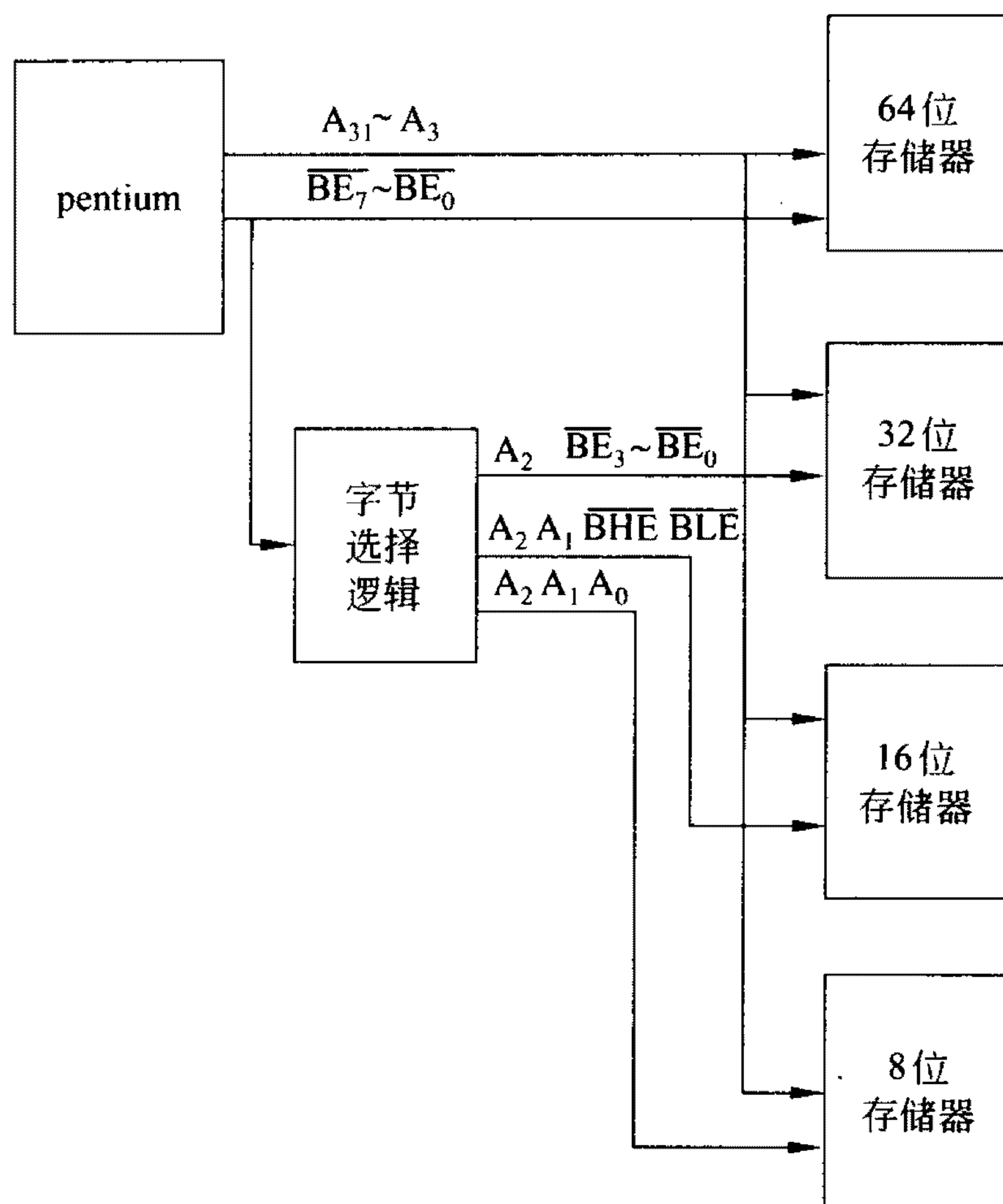


图 7-26 地址总线与 64 位、32 位、16 位和 8 位存储器的接口

习 题

1. 试说明微型计算机常用存储器的类型与特点。
2. 半导体存储器的性能指标有哪些？对微型计算机有何影响？
3. 存储器地址译码方式有几种？各有什么特点？
4. 动态 RAM 为什么要刷新？一般有几种刷新方式？各有什么优缺点？
5. 某 DRAM 存储芯片，其字位结构为 $1\text{M} \times 1\text{bit}$ ，试问其地址、数据引脚各是多少个？
6. 某 SRAM 存储芯片，其字位结构为 $512\text{K} \times 8\text{bit}$ ，试问其地址、数据引脚各是多少个？
7. 试用 6264 构成 32KB 的存储器，并画出相应的逻辑结构图。
8. 现有 $1024 \times 1\text{bit}$ 的存储芯片，若用它组成容量为 $16\text{K} \times 8\text{bit}$ 的存储器。试求：
 - (1) 实现该存储器所需的芯片数量。
 - (2) 该存储器所需的地址码总位数是多少？其中几位选片？几位用作片内地址？
9. 现有如下存储芯片： $2\text{K} \times 1\text{bit}$ 的 ROM； $4\text{K} \times 1\text{bit}$ 的 RAM； $8\text{K} \times 1\text{bit}$ 的 ROM，若用它们组成 16KB 的存储器，前 4KB 为 RAM，后 12KB 为 ROM，地址码采用 16 位。试问：

- (1) 各种存储芯片分别用多少片?
 - (2) 正确选用译码器及门电路,并画出相应的逻辑结构图。
 - (3) 指出有无地址覆盖现象。
10. 用 Intel 6116 芯片组成 8KB RAM, 设 CPU 的地址线为 16 根, ($A_0 \sim A_{15}$), 试问:
- (1) 需要几片 6116?
 - (2) 地址线 and 数据线各为多少根?
 - (3) 每一片的地址范围是多少? 是否有重叠区? (采用全译码法)
 - (4) 如何连线? (包括地址线、数据线和状态线)

输入/输出系统

8.1 概 述

8.1.1 接口电路

CPU 是通过接口电路与外部设备进行信息交换的。源程序或原始数据通过接口从输入设备输入,运行结果通过接口向输出设备输出。外设种类繁多,信号类型复杂,既有机械式、电动式、电子式,也有其他形式;所使用的信号,可以是数字量、模拟量(模拟式的电压、电流),也可以是开关量(两个状态的信息);信息处理的速度也不同,可以从键盘输入,也可以是磁盘输入;外设的数据传送方式可以是并行,也可以是串行。因此,在微型计算机和外设之间必须有输入/输出(I/O)接口设备,以使 CPU 与外设能够达到最佳匹配,实现高效、可靠的信息交换。

1. 接口电路应具备的功能

(1) 数据缓冲功能

接口电路中一般都设置数据缓冲器或锁存器,以解决高速主机与低速外设的矛盾,避免因处理速度不同而丢失信息。

(2) 联络功能

接口电路应能提供外设的状态。

(3) 寻址功能

接口电路应有 I/O 端口地址译码器,以产生片选信号或者是端口寄存器的选中信号。

(4) 数据转换功能

CPU 所处理的是并行数据,而有些外设只能处理串行数据,这时接口应具有数据“串→并”和“并→串”的转换功能。

(5) 中断管理功能

在高性能的接口电路中,为了便于 CPU 使用中断方式和端口寄存器交换信息,接口电路应设置中断控制电路,允许或禁止接口电路提出中断请求,而且中断控制电路的控制功能应交给 CPU,即 CPU 执行输出指令允许或禁止接口电路提出中断请求。

对于不同的外部设备,接口电路的功能也不相同。但前三项功能一般接口电路都具备。

2. 接口电路分类

接口按通用性可分为两类:专用接口和通用接口。专用接口即为某种用途或为某类外设而专门设计的接口电路,例如 CRT 显示控制器、键盘控制器、DMA 控制器等。通用接口是可供多种外部设备使用的标准接口,它可以连接各种不同的外设而不必增加附加电路。

接口按可编程性可分为两类:可编程接口和不可编程接口。可编程性是指在不改动硬件的情况下,用户只要修改初始化程序就可以改变接口的工作方式,增加了接口的灵活性和可扩充性。

接口按与外设数据的传送方式可分为:并行 I/O 接口和串行 I/O 接口。并行 I/O 接口与外设间的数据传送按字长传送(如:8 位或 16 位二进制数同时传送),串行 I/O 接口即接口与外设之间的数据传送是按位(一个二进制位)传送。

8.1.2 输入/输出端口

输入/输出接口电路通常都包含一组寄存器,这些能与 CPU 交换信息的寄存器称为 I/O 端口寄存器,简称“端口”。

在接口电路中,按端口寄存器存放信息的物理意义来分,端口可分为 3 类:数据端口、状态端口和控制端口。

1. 数据端口

数据端口存放数据信息。在输入过程中,数据信息由外设经过接口电路中的数据端口,到达系统的数据总线,被 CPU 读取。在输出过程中,数据信息由 CPU 输出,经过数据总线进入接口电路中的数据端口,再通过接口和外设间的数据线送到外设。

2. 状态端口

状态端口存放状态信息,即反映外设当前工作状态的信息,CPU 可读取这些信息,以查询外设当前的工作情况。对于输入接口电路,状态信息应能反映输入数据是否准备好;对于输出接口电路,状态信息应能反映输出设备的忙闲状态。

3. 控制端口

控制端口存放控制信息,控制信息是 CPU 通过接口传送给外设的,以控制外设工作,如控制输入输出装置启动或停止等。对于可编程接口电路,控制信息还负责选择可编程接口芯片的工作方式等。

状态信息、控制信息与数据信息是不同性质的信息,必须要分别传送。但在大部分微型计算机中,只有输入指令(IN)和输出指令(OUT)。因此,状态信息和控制信息也被广义地看成一种数据信息,即状态信息作为一种输入数据,而控制信息作为一种输出数

据,这样,状态信息和控制信息也通过数据总线来传送。为了区别输入数据和状态信息,数据端口和状态端口必须有不同的端口地址;为了区别输出数据和控制信息,数据端口和控制端口也必须有不同的端口地址,因此一个接口电路往往有好几个端口地址,CPU寻址的是端口寄存器,而不是笼统地寻址外设接口电路。

4. I/O 端口的编址方式

(1) 端口和存储单元统一编址

在这种方式中,把 I/O 端口作为存储器的一个单元来看待,故每个 I/O 端口占用存储器的一个地址。从输入端口输入一个数据,作为一次存储器读操作;而向输出端口输出一个数据,则作为一次存储器写操作。

其特点是:

- ① CPU 对外设的操作可使用存储器操作指令,不需要专门的输入/输出指令。
- ② 端口地址占用内存空间,使内存容量减少。
- ③ 执行存储器指令往往要比那些为独立的 I/O 而专门设计的指令慢。

(2) I/O 端口独立编址

在这种方式中,I/O 端口和存储器分别建立两个地址空间,单独编址。

其特点为:

- ① 对于 I/O 端口,CPU 须有专门的 I/O 指令去访问。
- ② 端口地址不占用内存空间。

在 PC 系列机中,I/O 端口采用独立编址方式。

8.1.3 输入/输出指令

1. I/O 端口与 CPU(AL 或 AX)的信息交换

(1) 直接寻址的输入/输出指令

当端口地址为一个字节时,可以采用直接寻址方式。采用直接寻址方式,最多可访问 256 个端口。

指令格式如下:

输入指令:

```
IN      AL,PORT      ;PORT 端口内容输入到 AL
IN      AX,PORT      ;PORT 端口和 PORT+1 端口内容输入到 AX
IN      EAX,PORT     ;PORT~PORT+3 端口内容输入到 EAX
```

输出指令:

```
OUT     PORT,AL      ;AL 内容输出到 PORT 端口
OUT     PORT,AX      ;AX 内容输出到 PORT 端口和 PORT+1 端口
OUT     PORT,EAX     ;EAX 内容输出到 PORT~PORT+3 端口
```

(2) DX 间址的输入/输出指令

端口地址为两个字节时,用间接寻址方式,此时最多可寻址 2^{16} 个端口,而且端口地址必须放在寄存器 DX 中,其指令格式为:

输入指令：

IN AL,DX ;从 DX 指向的端口中读一个字节到 AL
 IN AX,DX ;从 DX 和 DX+1 指向的 2 个端口读一个字到 AX
 IN EAX,DX ;从 DX~DX+3 指向的 4 个端口读一个双字到 EAX

输出指令：

OUT DX,AL ;将 AL 内容输出到 DX 指向的端口
 OUT DX,AX ;将 AX 内容输出到 DX 指向的端口
 OUT DX,EAX ;将 EAX 中的双字输出到 DX~DX+3 指向的 4 个端口

2. I/O 端口与 RAM 单元信息交换

(1) 基本型格式

① 字节输入指令 INSB

功能：从 DX 间址的 I/O 端口取一个字节→ES: [DI]字节型 RAM 单元。若方向标志 D=0,则自动完成 DI+1→DI;若 D 标志=1,则自动完成 DI-1→DI。

② 字节输出指令 OUTSB

功能：从 DS: [SI]字节型单元取一个字节→DX 间址的 I/O 端口。若方向标志 D=0,则自动完成 SI+1→SI;若 D 标志=1,则自动完成 SI-1→SI。

③ 字输入指令 INSW

功能：从 DX 和 DX+1 间址的 I/O 端口取一个字→ES: [DI]字型 RAM 单元。若方向标志 D=0,则自动完成 DI+2→DI;若 D 标志=1,则自动完成 DI-2→DI。

④ 字输出指令 OUTSW

功能：从 DS: [SI]字型单元取一个字→DX、DX+1 间址的 I/O 端口。若方向标志 D=0,则自动完成 SI+2→SI;若 D 标志=1,则自动完成 SI-2→SI。

(2) 有重复前缀的格式

① 字节输入指令 REP INSB

② 字节输出指令 REP OUTSB

③ 字输入指令 REP INSW

④ 字输出指令 REP OUTSW

功能：同基本型格式。有“REP”前缀,表示每完成一次字节或字传送都执行 CX-1→CX 的操作,直到 CX=0 为止。

8.2 微型计算机系统与输入/输出设备的信息交换

微型计算机系统与输入/输出设备的信息交换有无条件传送方式、查询方式、中断控制方式和存储器直接存取(DMA)方式。

8.2.1 无条件传送方式

无条件传送方式具有以下特点：假设外设已准备好,即输入数据已准备好,或输出设

备空闲,此时 CPU 可以直接用 IN 或 OUT 指令完成与接口之间的数据传送。

采用无条件传送方式的接口电路,如图 8-1 和图 8-2 所示。

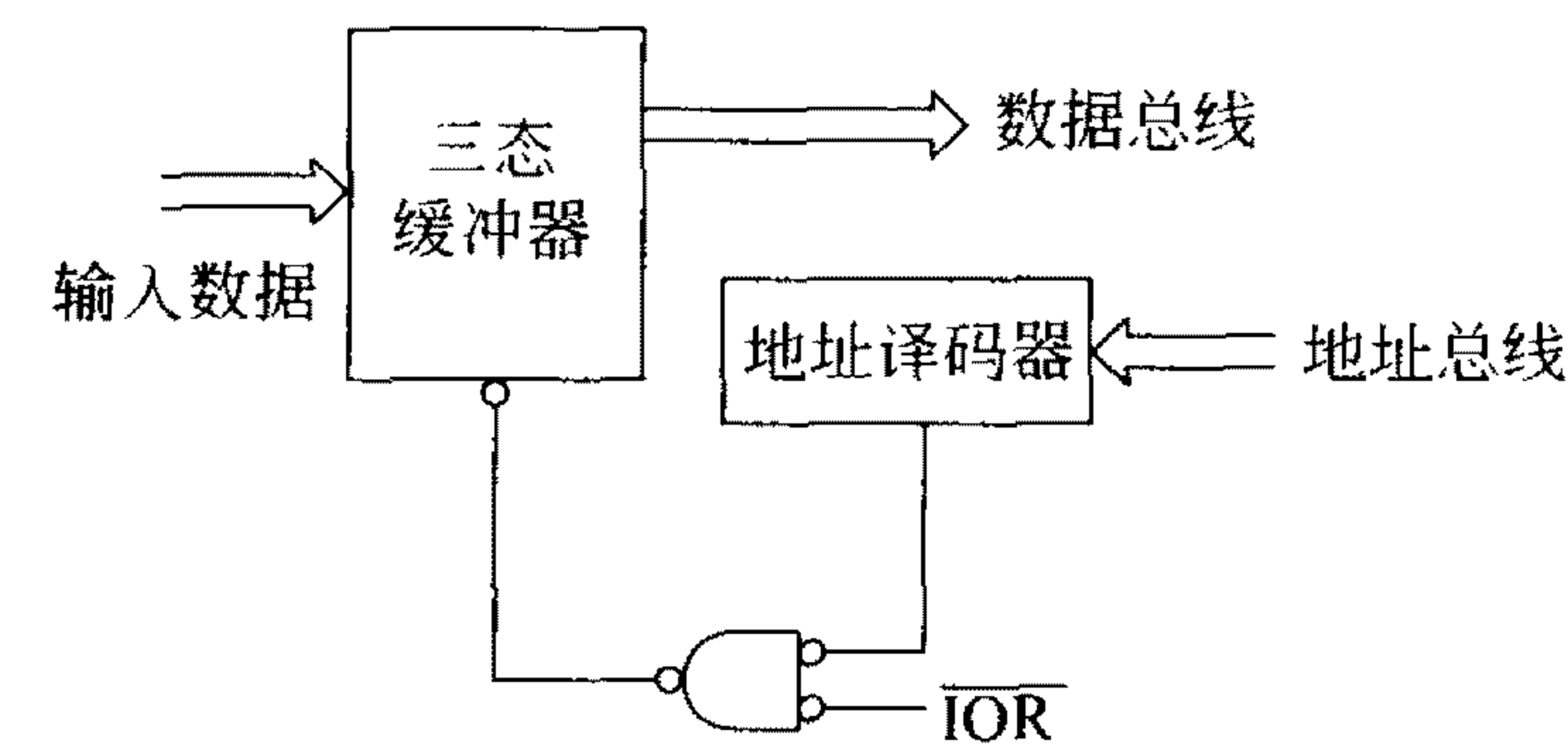


图 8-1 无条件传送的输入方式

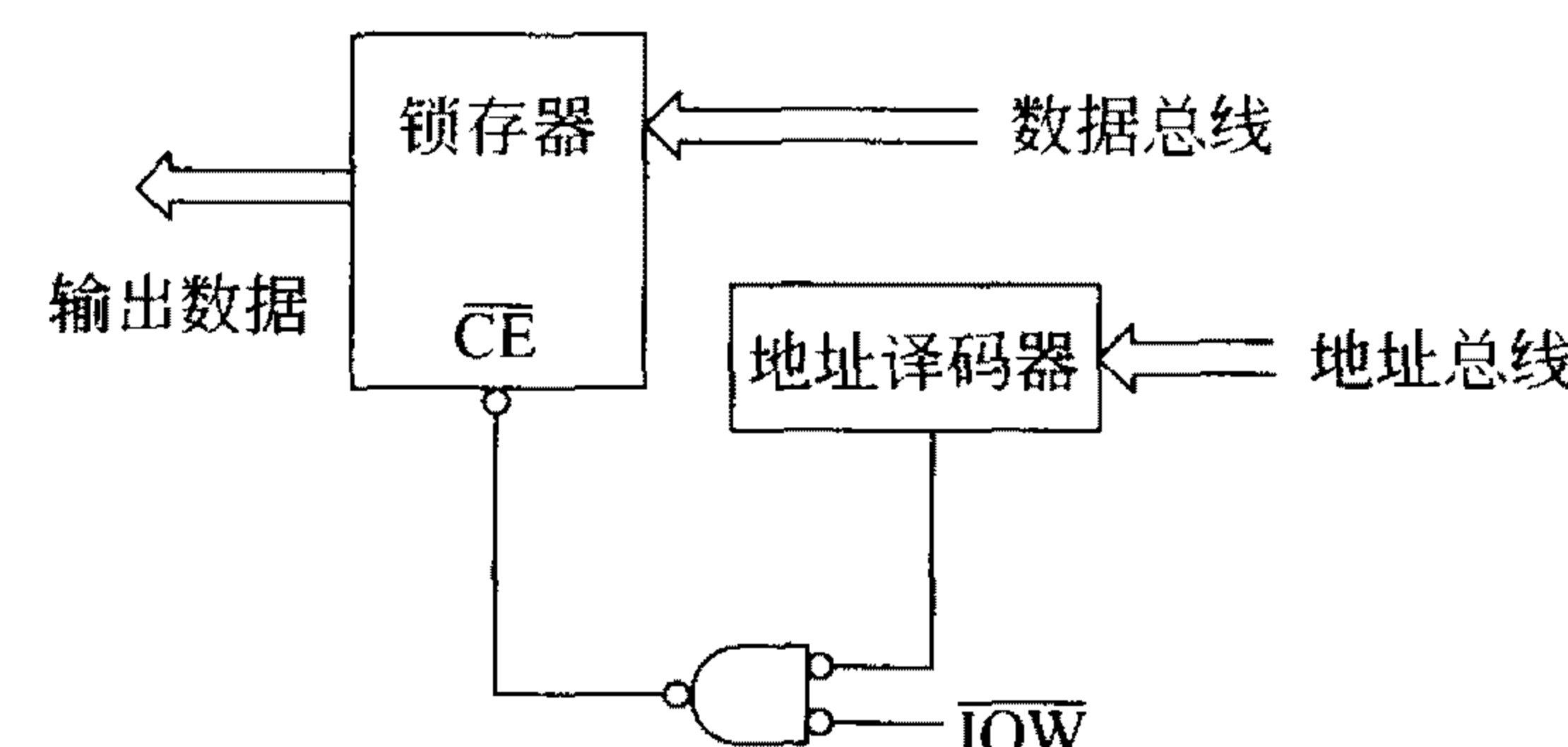


图 8-2 无条件传送的输出方式

无条件输入时,输入端可用三态缓冲器与 CPU 的数据总线相连。当 CPU 执行输入指令时,外设的数据已经准备好,I/O 读信号 \overline{IOR} 有效,输入数据通过三态缓冲器到达数据总线,供 CPU 读取。

无条件输出时,由于外设速度较慢,输出端与锁存器相连。CPU 执行输出指令时,必须保证锁存器是空闲的,I/O 写信号 \overline{IOW} 有效,CPU 输出的数据由数据总线送入输出锁存器,输出锁存器一直保存这个数据,直到被外设取走。

8.2.2 查询方式

采用查询方式接收数据前,CPU 要查询输入数据是否准备好;采用查询方式输出数据前,CPU 要查询输出设备是否空闲。只有确认外设已具备了输入或输出条件后,才能用 IN 或 OUT 指令完成数据传送。

和无条件传送方式相比,查询方式的接口电路中要设置供 CPU 查询的电路。

1. 查询式输入

图 8-3 为查询式输入接口电路,该电路有两个端口寄存器,即状态端口和数据端口。

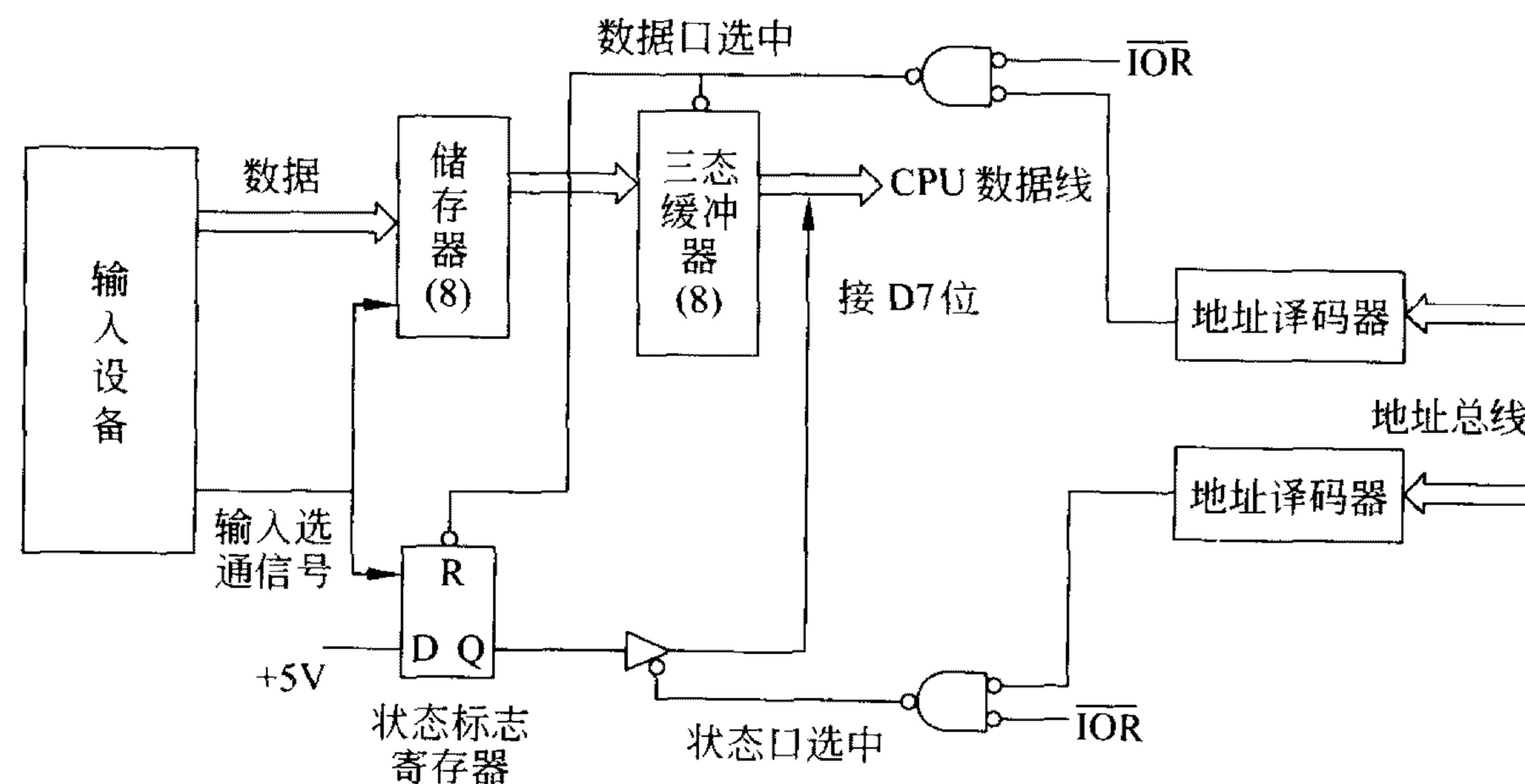


图 8-3 查询式输入接口电路

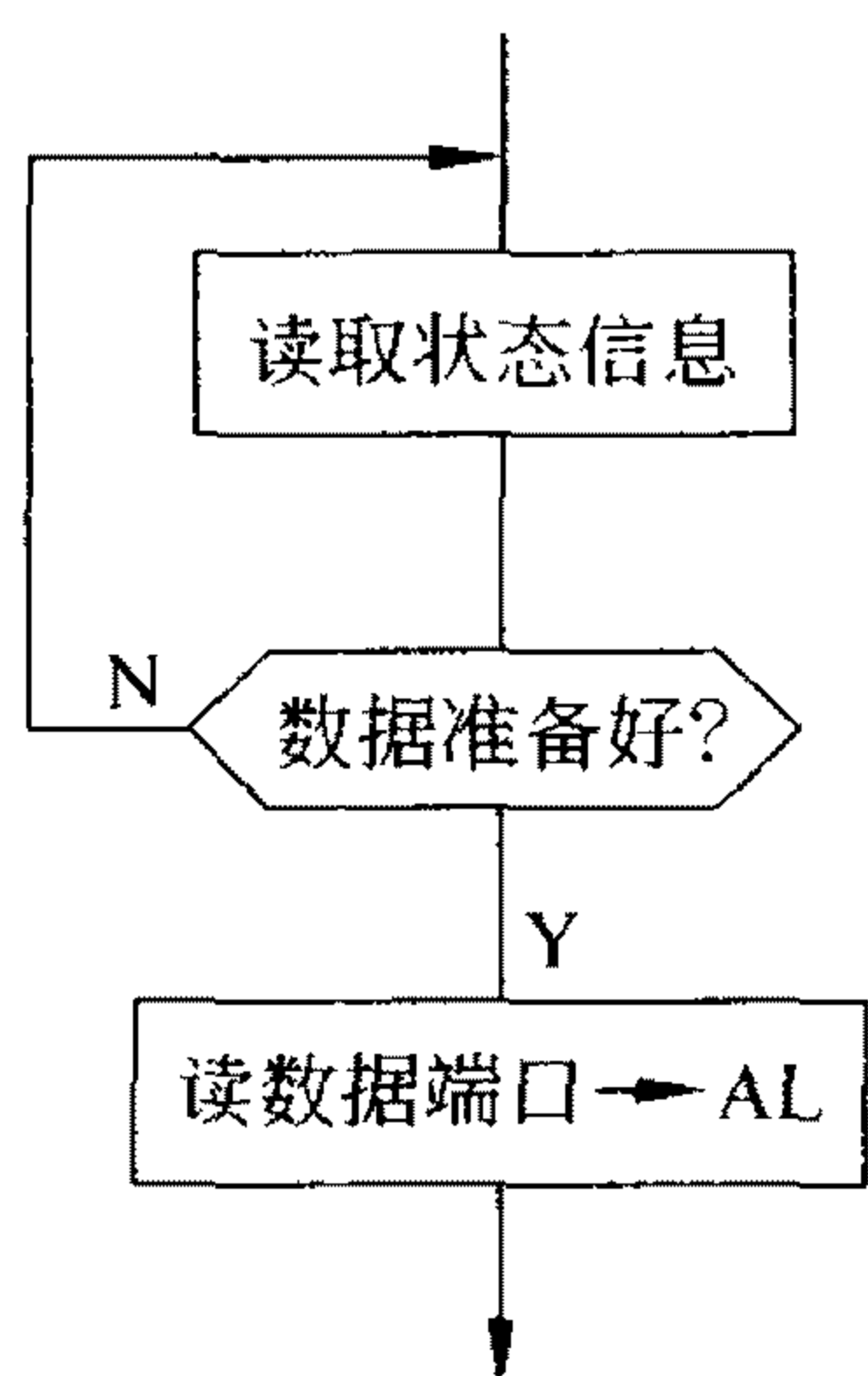


图 8-4 查询式输入程序流程图

输入设备准备好数据后,发出输入选通信号,一方面把数据送入数据锁存器,另一方面使状态标志触发器置 1,状态标志是一位信息,接到 CPU 数据线的某一位上,假设接至 D_7 位。CPU 先读取状态端口,查询 D_7 位是否为 1,若是,表示输入数据已准备好,然后读取数据端口,取走输入数据,同时将状态标志触发器复位。图 8-4 为查询式输入程序流程图。

查询式输入的程序段如下:

```

SCAN:  IN      AL, 状态端口地址
        TEST    AL, 80H
        JZ      SCAN
        IN      AL, 数据端口地址
  
```

2. 查询式输出

图 8-5 为查询式输出接口电路,图中“状态端口”和“数据端口”合用一个端口地址。

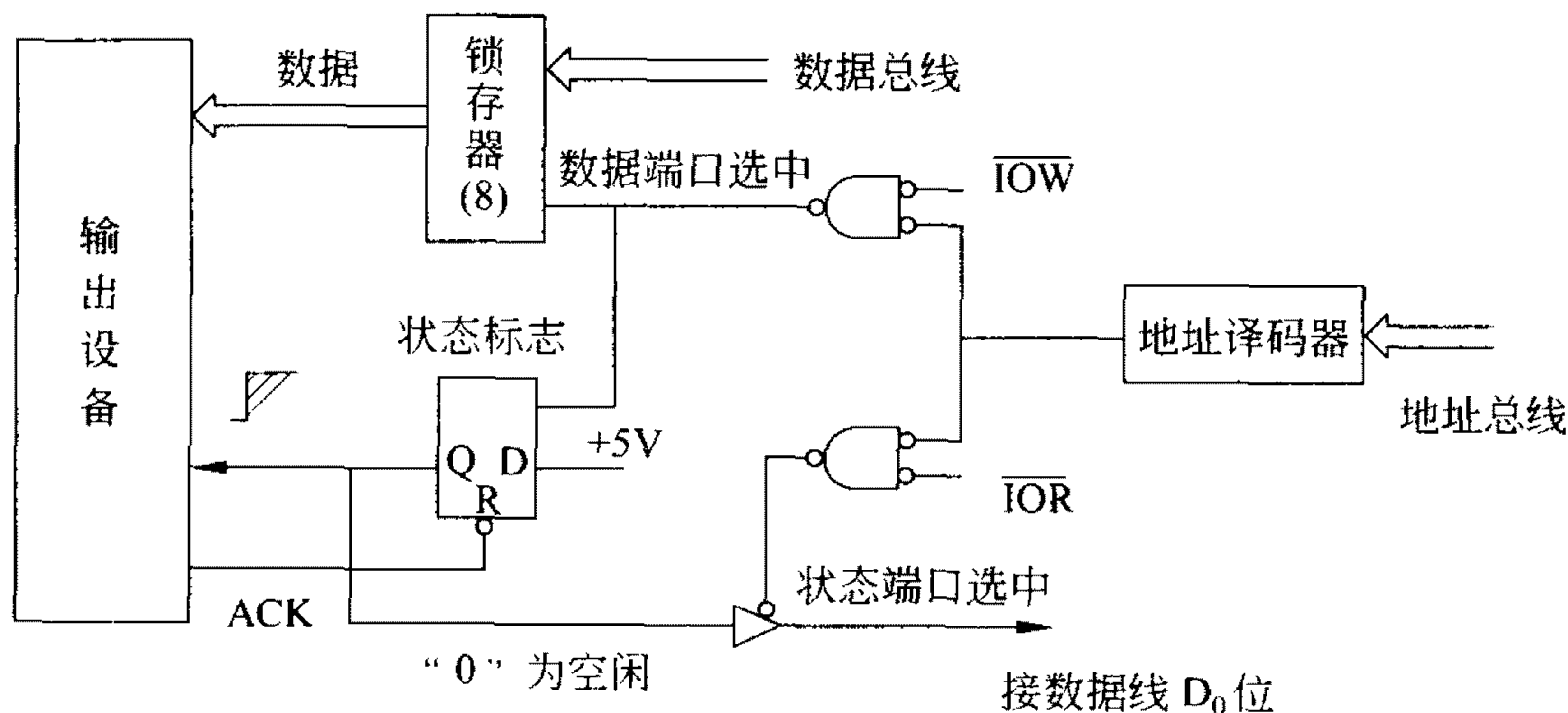


图 8-5 查询式输出接口电路

输出设备空闲时,状态标志触发器置 0,输出数据前,CPU 先读取状态信息,假设忙闲标志接至数据线的 D_0 位,当 D_0 位为 0 时,表示输出设备空闲,然后 CPU 对数据端口执行输出指令,“数据端口选中”信号一方面把输出数据写入锁存器,一方面使状态标志触发器置 1,通知输出设备。输出设备取走当前数据后,向接口回送“确认”(ACK)信号,使状态标志触发器置 0,表示输出设备空闲。

图 8-6 为查询式输出流程图。

查询式输出程序段如下:

```

SCAN:  IN      AL, 状态端口地址      ;取状态信息
        TEST    AL, 01H              ;测忙闲标志
        JNZ     SCAN                 ;忙,转移
        MOV     AL, 某数
        OUT     数据端口地址, AL     ;空闲,输出数据
  
```

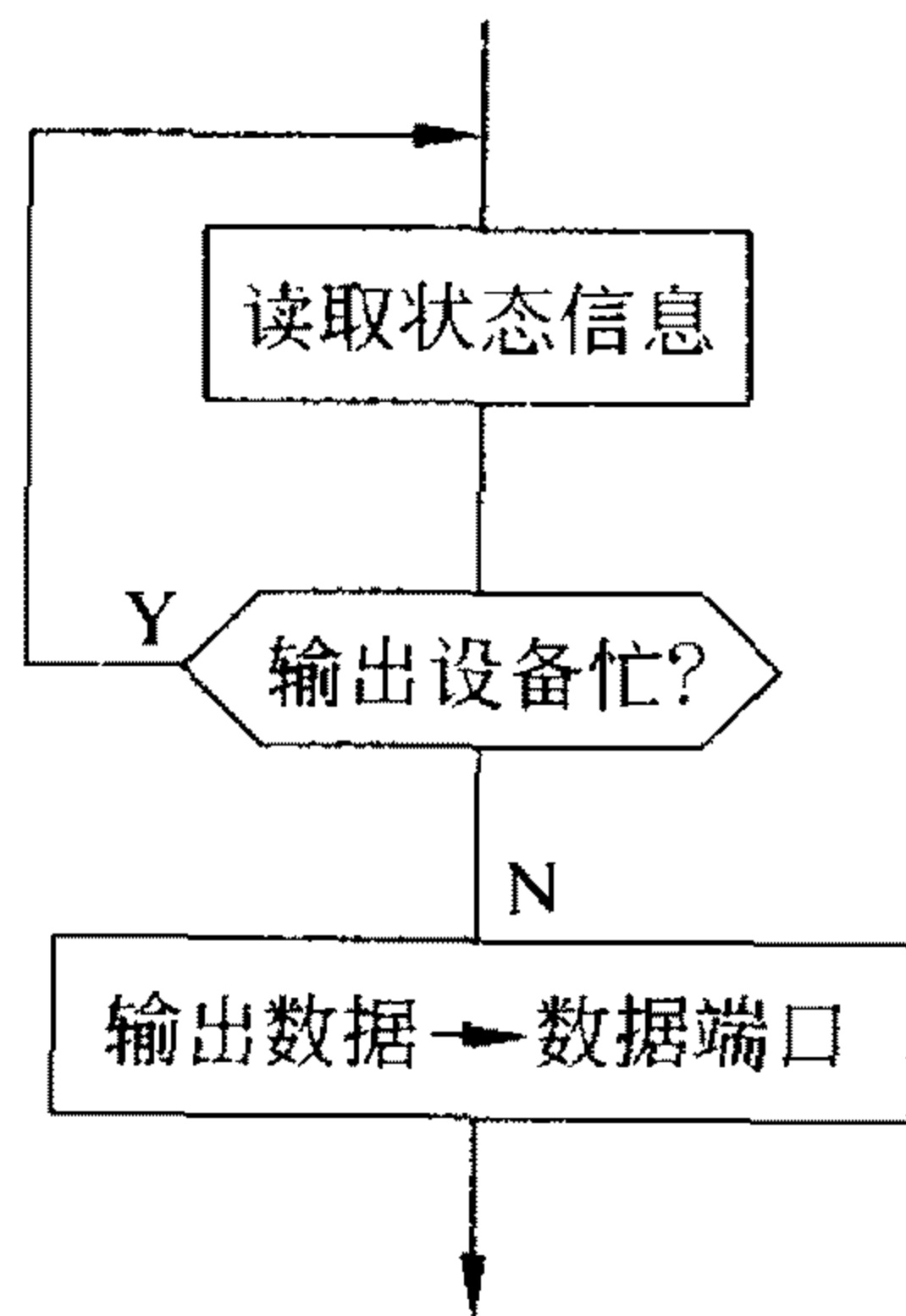


图 8-6 查询式输出流程图

8.2.3 中断控制方式

在查询方式中,CPU 通过不断地读取状态信息来了解外设状态,CPU 利用率不高;另外采用查询方式工作,不能保证系统实时地对外设的请求作出响应。为了提高 CPU 的效率,使系统具有实时性能,产生了中断处理技术。采用中断方式传送信息时,如果外设未做好数据传送的准备,CPU 可执行与传送数据无关的其他指令;当外设做好传送准备后,可向 CPU 发出中断请求,请求 CPU 为之服务;若 CPU 响应中断请求,将暂停正在运行的程序,转入中断服务程序,完成数据的传送;等中断服务结束后,将自动返回原来运行的程序继续执行。

图 8-7 为中断方式的输入接口示意图,图中数据端口和中断控制端口合用一个端口地址。

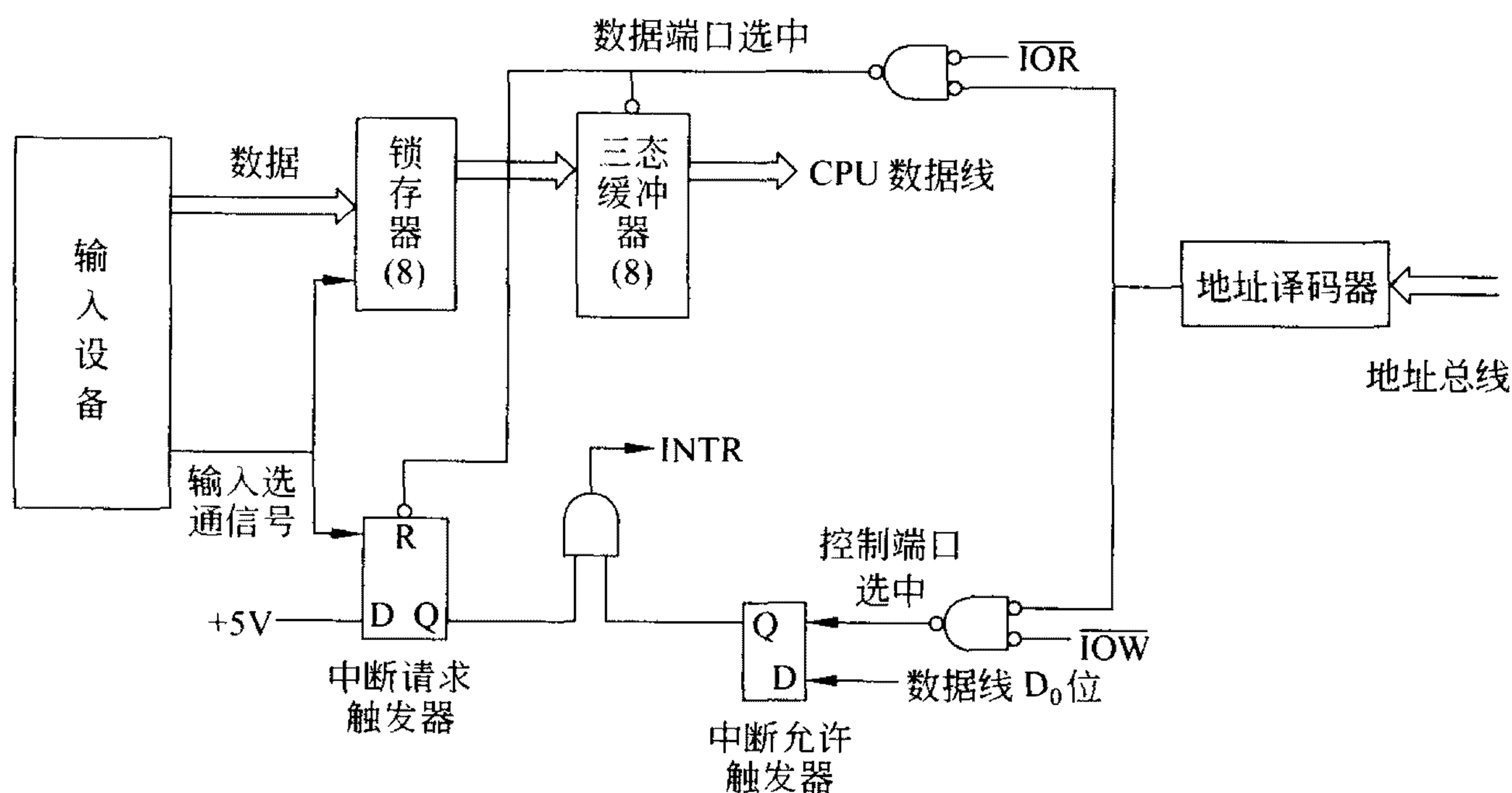


图 8-7 中断方式输入接口电路

当输入数据准备好后,发出输入选通信号,将数据写入锁存器中,同时将中断请求触发器置 1,向 CPU 提出中断请求。CPU 响应中断后,转而执行中断服务程序,在服务程序中,执行输入指令,选中数据端口,一方面强令中断请求触发器复位,另一方面打开三态缓冲器,把锁存器中的数据送到 CPU 数据线上,完成一次数据输入操作。

在中断方式的接口电路中,为了增强中断的灵活性,一般设置中断允许触发器,该触发器受 CPU 控制,如图 8-7 所示。当向端口写入 01H 时,中断允许触发器置 1,这时,如果输入数据准备好,接口就可以发出中断请求了;如果向端口写入 00H,则中断允许触发器置 0,禁止中断请求。

有关中断方式的详细内容请见后续章节。

8.2.4 直接存储器存取方式

采用中断方式进行数据传送,可以提高 CPU 的利用率。但是,中断传送是由 CPU 通过程序来实现的,每次执行中断服务程序需要保护断点,在中断服务程序中,需要保护

现场,为中断源服务,中断服务结束还需要恢复现场,CPU 需要执行若干指令来完成上述工作。对于高速外设,如高速磁盘驱动器或高速数据采集系统来说,中断方式往往不能满足要求。

直接存储器存取方式是用硬件实现在外设与内存之间直接进行数据交换,而不是通过 CPU 间接交换,这样数据传送速度的上限就取决于存储器的工作速度。这种方式称为直接存储器存取(Direct Memory Access,DMA)方式,为实现 DMA 方式而设计的专用控制芯片,称为 DMA 控制器(DMAC)。例如,Intel 公司的 8237A 等。

DMA 方式的优点是传送速度快,这是由于 CPU 不参与操作,因此省去了 CPU 取指令,指令译码,存取数据等操作。DMA 方式的主要缺点是:硬件电路比较复杂。

有关 DMA 的详细论述,请见后续章节。

8.3 可编程定时器/计数器 8254

计算机系统中需要用到定时信号。比如:动态存储器的刷新定时、系统日时钟的计时以及发声系统的声源等。

一般定时操作可用软件和硬件两种方法实现。用软件方法实现定时,是通过执行延时程序来实现的,这种方法不需要硬件设备,但是,CPU 执行延时程序将增加时间开销,降低了 CPU 的效率。用硬件方法实现定时一般采用定时器/计数器,可编程定时器/计数器具备定时和计数两个功能。

常用的可编程定时器/计数器有 8253 和 8254 等,8254 是 8253 的增强型,它具备 8253 的全部功能,凡是应用 8253 的系统,均可用 8254 取代。在高档微型计算机系统中,定时器/计数器常由多功能芯片实现,但在性能上与 8253/8254 兼容。

8.3.1 8254 的内部结构

8254 内部有 3 个独立的 16 位计数器,每个计数器有 6 种工作方式,计数初值的数制可设定为二进制或 BCD 码,每个计数器允许的最高计数频率为 10MHz,有读出命令。

8254 的内部结构框图如图 8-8 所示,它由与 CPU 的接口、内部控制电路及 3 个计数器构成。

1. 数据总线缓冲器

数据总线缓冲器是一个三态、双向 8 位寄存器,用于将 8254 与系统总线 $D_7 \sim D_0$ 相连。数据总线缓冲器有 3 个基本功能:CPU 通过数据总线缓冲器向 8254 写入确定工作方式的命令字;向某一计数器写入计数初值;从某一计数器读取当前的计数值。

2. 读/写逻辑

读/写逻辑为 8254 内部的控制电路,当片选信号 $\overline{CS}=0$ 时,由 A_1, A_0 信号(通常接

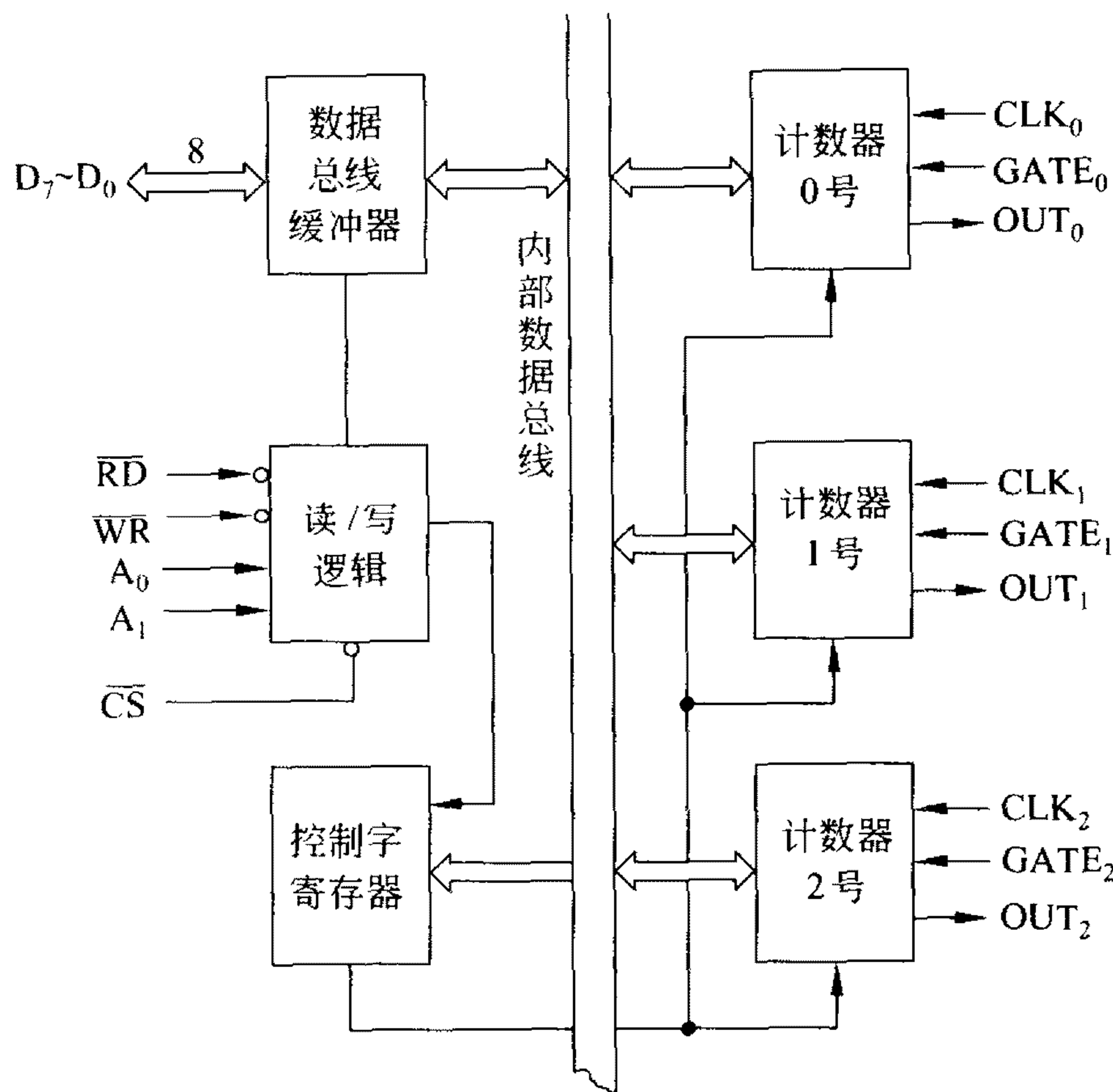


图 8-8 8254 的内部结构框图

CPU 地址线 A_1, A_0) 选择内部寄存器, 由读信号 \overline{RD} (通常接 CPU 的 \overline{IOR}) 和写信号 \overline{WR} (通常接 CPU 的 \overline{IOW}) 完成对选定寄存器的读/写操作。

当片选信号 $\overline{CS}=1$ 时, 数据总线缓冲器与系统数据总线脱开。

3. 控制字寄存器

初始化编程时, 由 CPU 写入控制字, 以决定计数器的工作方式, 设置读出命令。此寄存器只能写入, 不能读出。

4. 计数器

8254 有 3 个独立的计数器, 每个计数器的结构完全相同, 如图 8-9 所示。

每个计数器对外有 3 个引脚: $GATE_i$ 为门控信号输入端, CLK_i 为计数脉冲输入端, OUT_i 为输出信号端。

初始化编程时, 程序员向计数初值寄存器写入的计数初值 (只要不写入新的初值, 该值始终保持不变), 将自动送入 16 位减 1 计数器。当 $GATE_i=1$ 时, 每一个 CLK_i 信号的下降沿使减 1 计数器减 1, 当计数值减到某个规定数值时 (取决于设定的工作方式), OUT_i 端产生输出信号。在计数过程中, 锁存器随减 1 计数器的变化而变化。

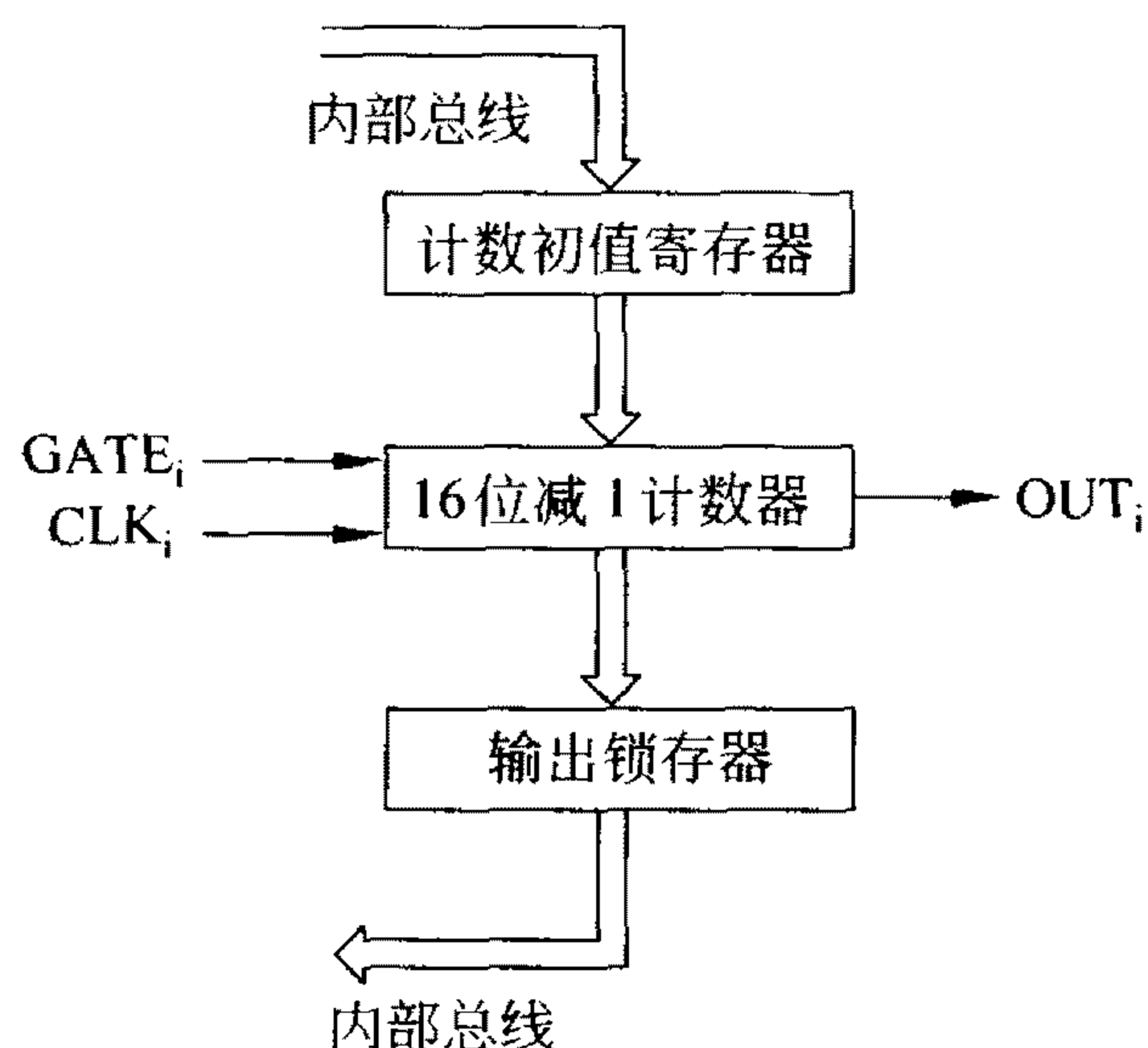


图 8-9 计数器结构示意图

计数脉冲可以是有规律的时钟信号,用于定时;也可以是随机脉冲信号,用于计数。

计数初值 N 的计算公式如下:

$$N = f_{\text{CLK}_i} \div f_{\text{OUT}_i}$$

5. 8254 端口地址

在 $\overline{\text{CS}}$ 等于 0 的前提下:

$A_1 A_0 = 00$, 选中 0 号计数器;

$A_1 A_0 = 01$, 选中 1 号计数器;

$A_1 A_0 = 10$, 选中 2 号计数器;

$A_1 A_0 = 11$, 选中控制字寄存器。

8.3.2 8254 引脚功能

8254 的外部引脚,如图 8-10 所示。

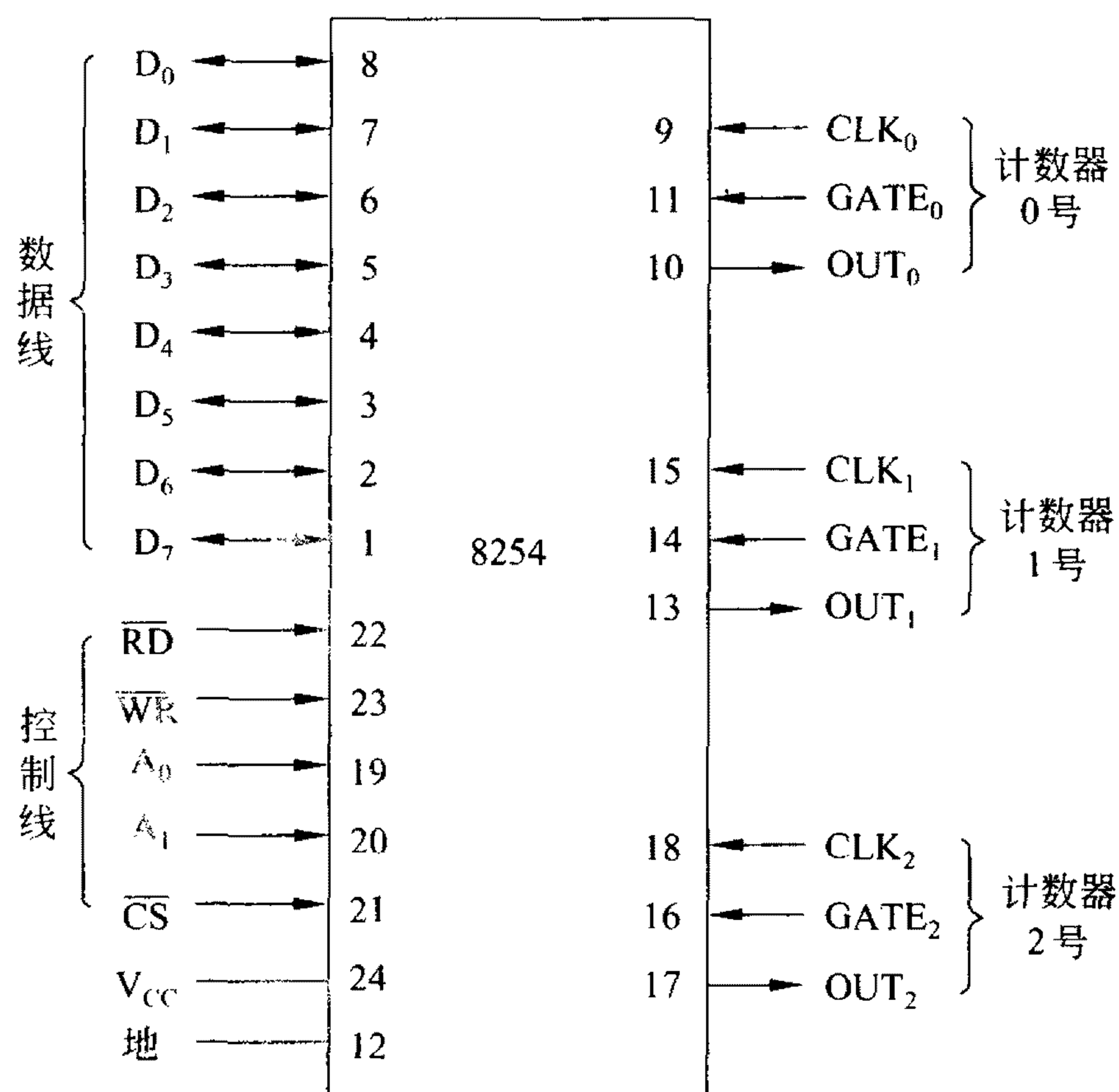


图 8-10 8254 外部引脚图

8254 使用单一 +5V 电源,有 24 个引脚,采用双列直插式封装。

$D_7 \sim D_0$ 为数据线,与 CPU 数据线相连。 $\overline{\text{CS}}$ 为片选信号输入端。 A_1 、 A_0 为内部寄存器选择信号,接 CPU 地址线 A_1 、 A_0 。 $\overline{\text{RD}}$ 、 $\overline{\text{WR}}$ 接收来自 CPU 的输入、输出读/写命令。 $\text{GATE}_0 \sim \text{GATE}_2$ 、 $\text{CLK}_0 \sim \text{CLK}_2$ 和 $\text{OUT}_0 \sim \text{OUT}_2$ 是 3 个计数器的外部引脚。

表 8-1 给出了 8254 内部寄存器的读/写操作。

表 8-1 8254 内部寄存器读/写操作

$\overline{\text{CS}}$	$\overline{\text{RD}}$	$\overline{\text{WR}}$	A_1	A_0	操 作
0	1	0	0	0	计数初值写入 0 号计数器
0	1	0	0	1	计数初值写入 1 号计数器
0	1	0	1	0	计数初值写入 2 号计数器
0	1	0	1	1	向控制字寄存器写控制字
0	0	1	0	0	读 0 号计数器当前计数值
0	0	1	0	1	读 1 号计数器当前计数值
0	0	1	1	0	读 2 号计数器当前计数值
0	0	1	1	1	无操作
1	×	×	×	×	禁止
0	1	1	×	×	无操作

8.3.3 8254 的工作方式

8254 的 3 个计数器均有 6 种工作方式,其主要区别在于:

- ① 输出波形不同;
- ② 启动计数器的触发方式不同;
- ③ 计数过程中门控信号 GATE 对计数操作的影响不同;
- ④ 有的工作方式具备“初值自动重装”的功能。初值自动重装的功能是:当计数值减到规定的数值后,计数初值将会自动地装入计数器重新进行计数。

1. 方式 0——计数结束输出正跃变信号

8254 工作在方式 0 时,其工作波形如图 8-11 所示。

方式 0 工作的特点是:

- ① 写入控制字后,OUT 端输出低电平,写入计数初值后,OUT 端保持低电平,计数器开始对 CLK 脉冲进行减 1 计数。当计数值减为 0 时,OUT 端输出变为高电平。此信号可用于向 CPU 发出中断请求。

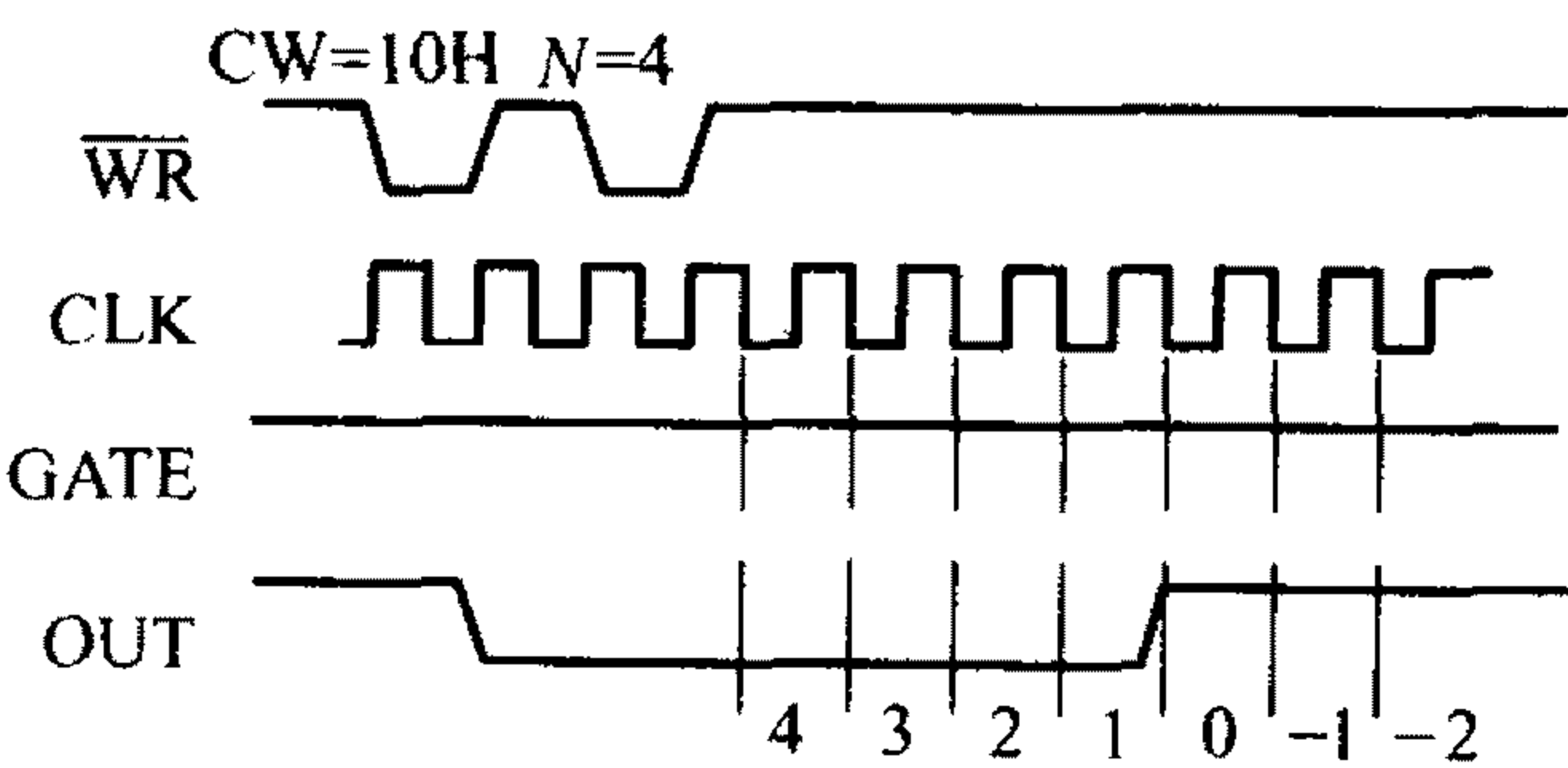


图 8-11 方式 0 波形图

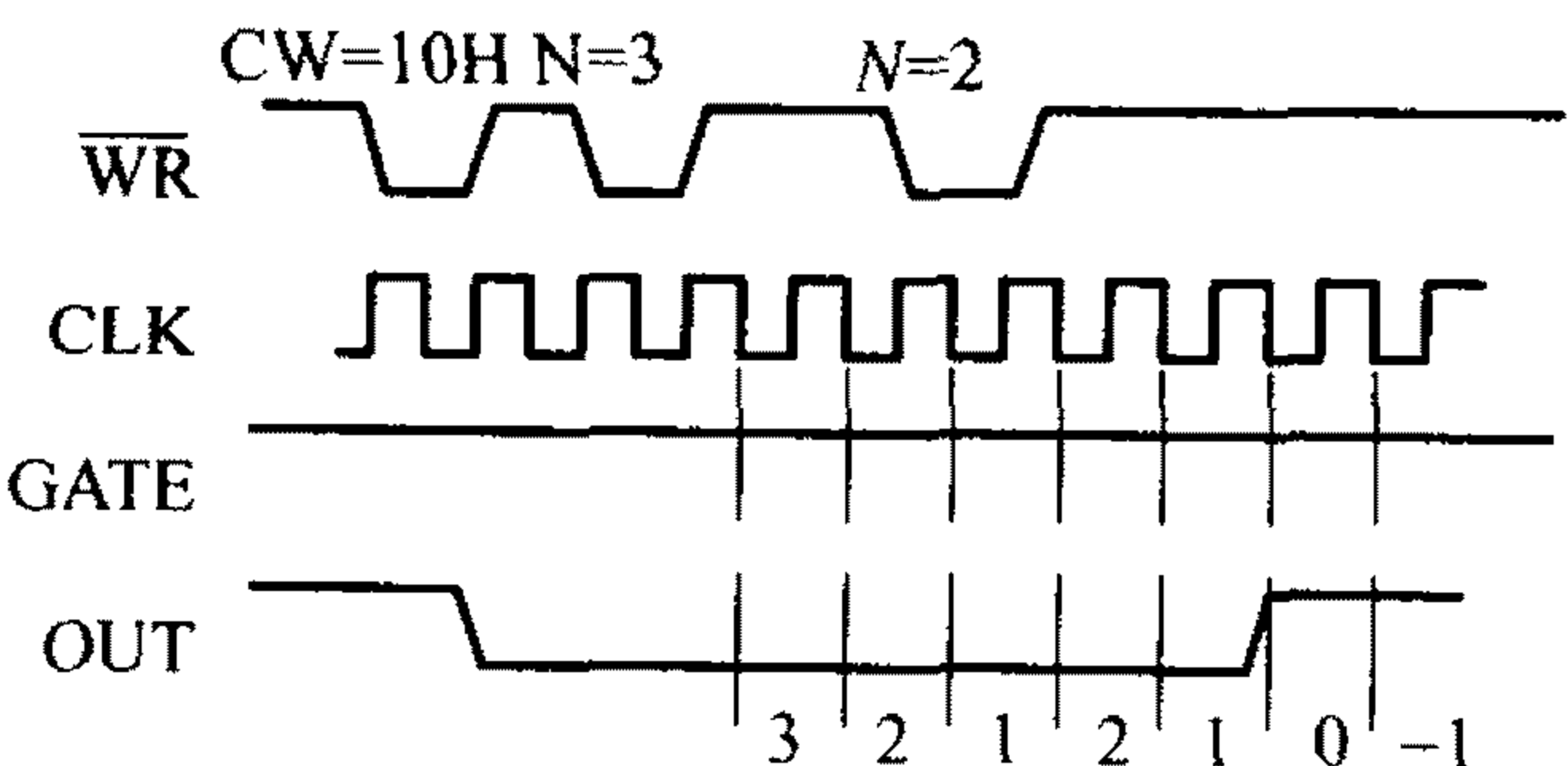


图 8-12 方式 0 计数过程中改变计数初值

方式 0 不具备“初值自动重装”的功能。

- ② 在计数过程中,如果改变计数初值,则在写入新的计数初值后,计数器将以新的值为计数初值,重新开始减 1 计数。如图 8-12 所示。

③ GATE 为计数控制信号,当 GATE=1 时,允许计数;GATE=0 时,停止计数。其波形如图 8-13 所示。

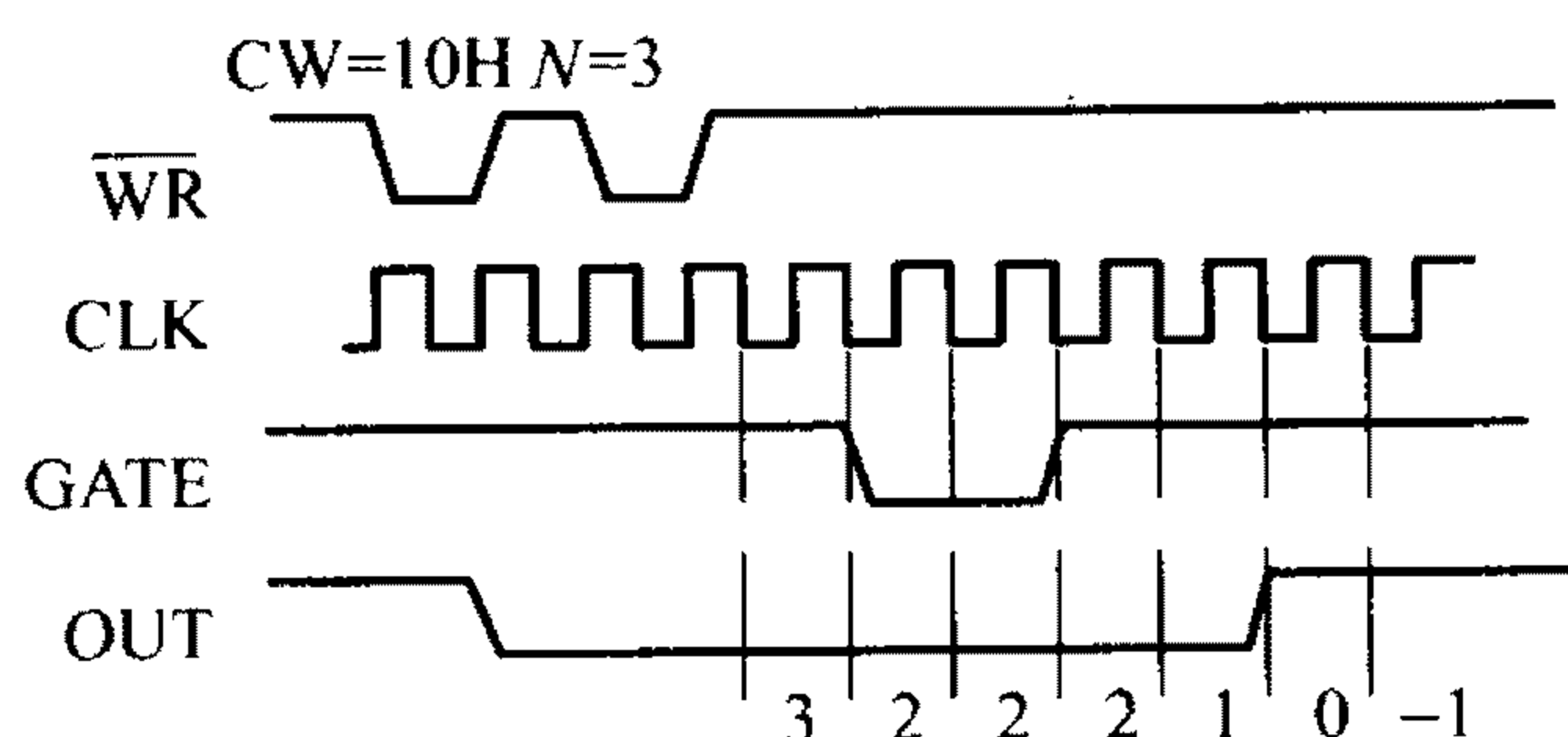


图 8-13 方式 0 时 GATE 信号的作用

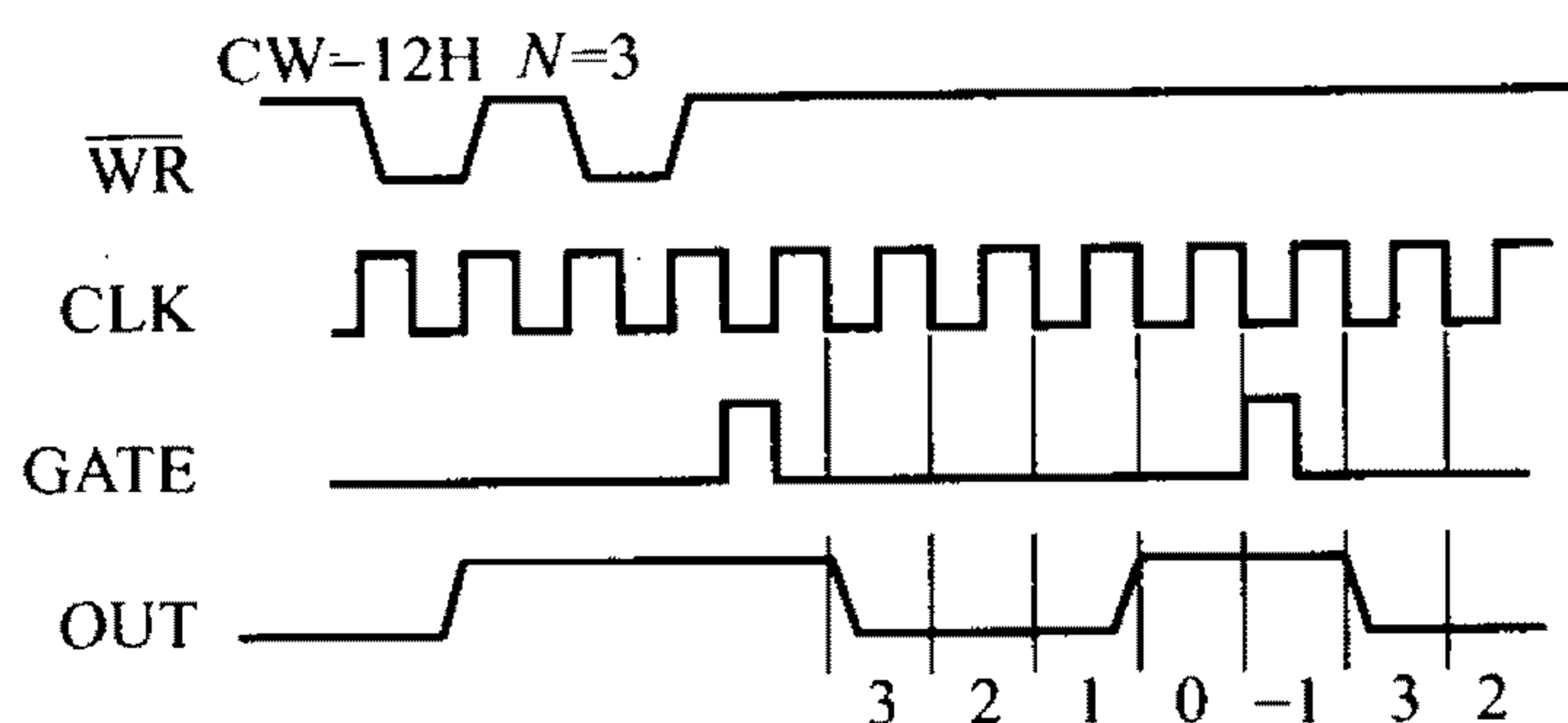


图 8-14 方式 1 波形图

2. 方式 1——单脉冲发生器

8254 工作在方式 1 时,其工作波形如图 8-14 所示。

方式 1 是由外部门控脉冲(硬件)启动计数。其特点是:

① 写入控制字后,OUT 端输出高电平,写入计数初值后,OUT 端保持高电平,计数器由 GATE 的上升沿启动。GATE 启动之后,OUT 变为低电平,每来一个 CLK 脉冲,计数器减 1,当计数值减到 0 时,OUT 输出高电平,在 OUT 端输出一个负脉冲,负脉冲宽度为计数初值乘以 CLK 脉冲周期。

② 在计数器未减到 0 时,如果门控信号 GATE 又来一个正脉冲,计数初值将重新装入计数器,计数器从初始值开始重新作减 1 计数。如图 8-15 所示。

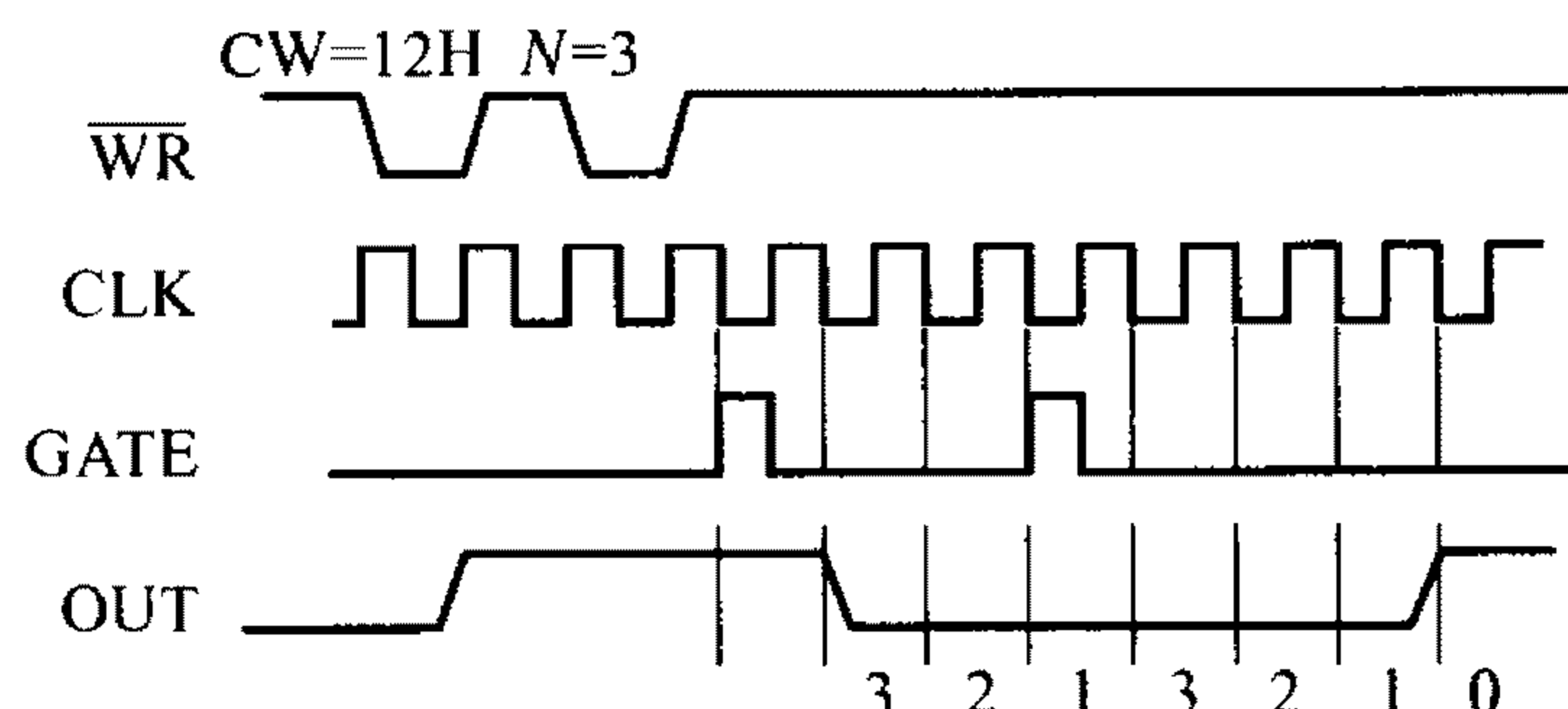


图 8-15 方式 1 时 GATE 信号的作用

③ 在计数过程中,程序员可装入新的计数初值,但计数过程不受影响。只有当 GATE 再次出现 0→1 的跃变后,计数器才能按新的计数初值作减 1 计数。如图 8-16 所示。

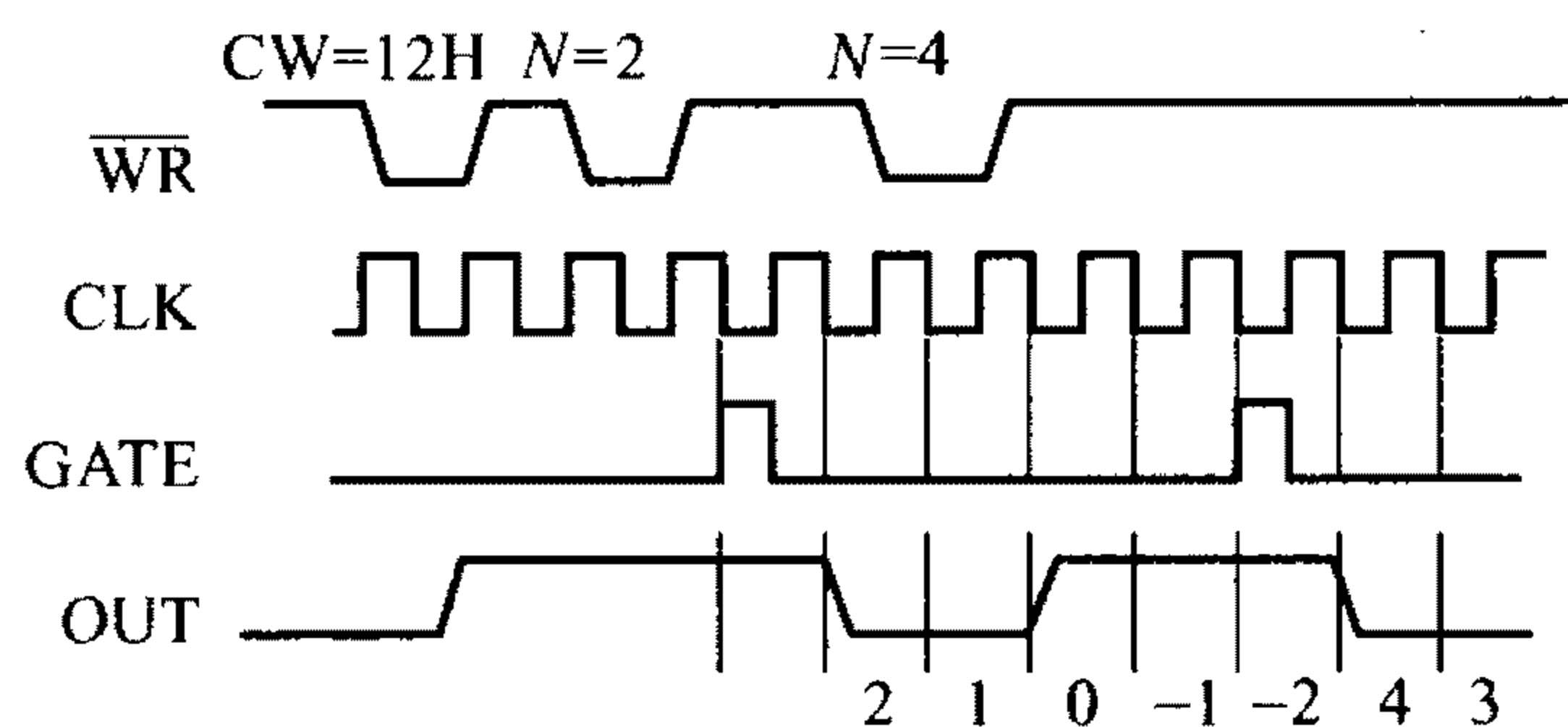


图 8-16 方式 1 在计数过程中改变计数初值

3. 方式 2——分频器

8254 工作在方式 2 时,其工作波形如图 8-17 所示。

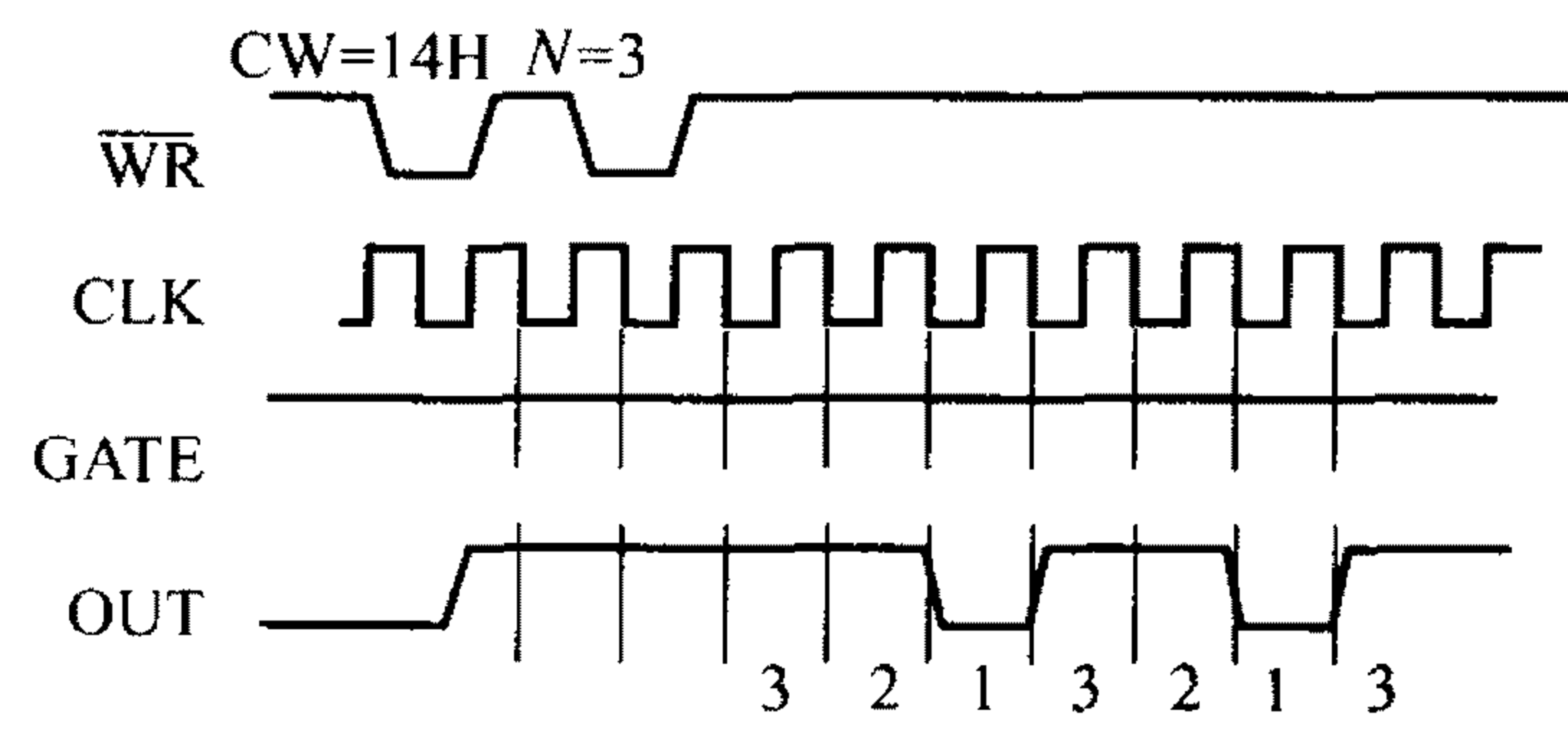


图 8-17 方式 2 波形图

方式 2 的特点是计数器有“初值自动重装”的功能,输出波形为周期脉冲。

其工作特点如下:

- ① 写入控制字后,OUT 输出为高电平,写入计数初值后,如果 GATE 为高电平,计数器开始减 1 计数,当计数值减到 1 时,OUT 输出低电平,经过一个 CLK 周期,又变为高电平,并且计数初值自动重装,计数器开始重新计数,周而复始。OUT 端输出是连续的负脉冲,负脉冲宽度为一个 CLK 周期。
- ② 如果在减 1 计数过程中,GATE 变低,则暂停计数,GATE 的上升沿使计数器恢复初值,并从初值开始减 1 计数,如图 8-18 所示。

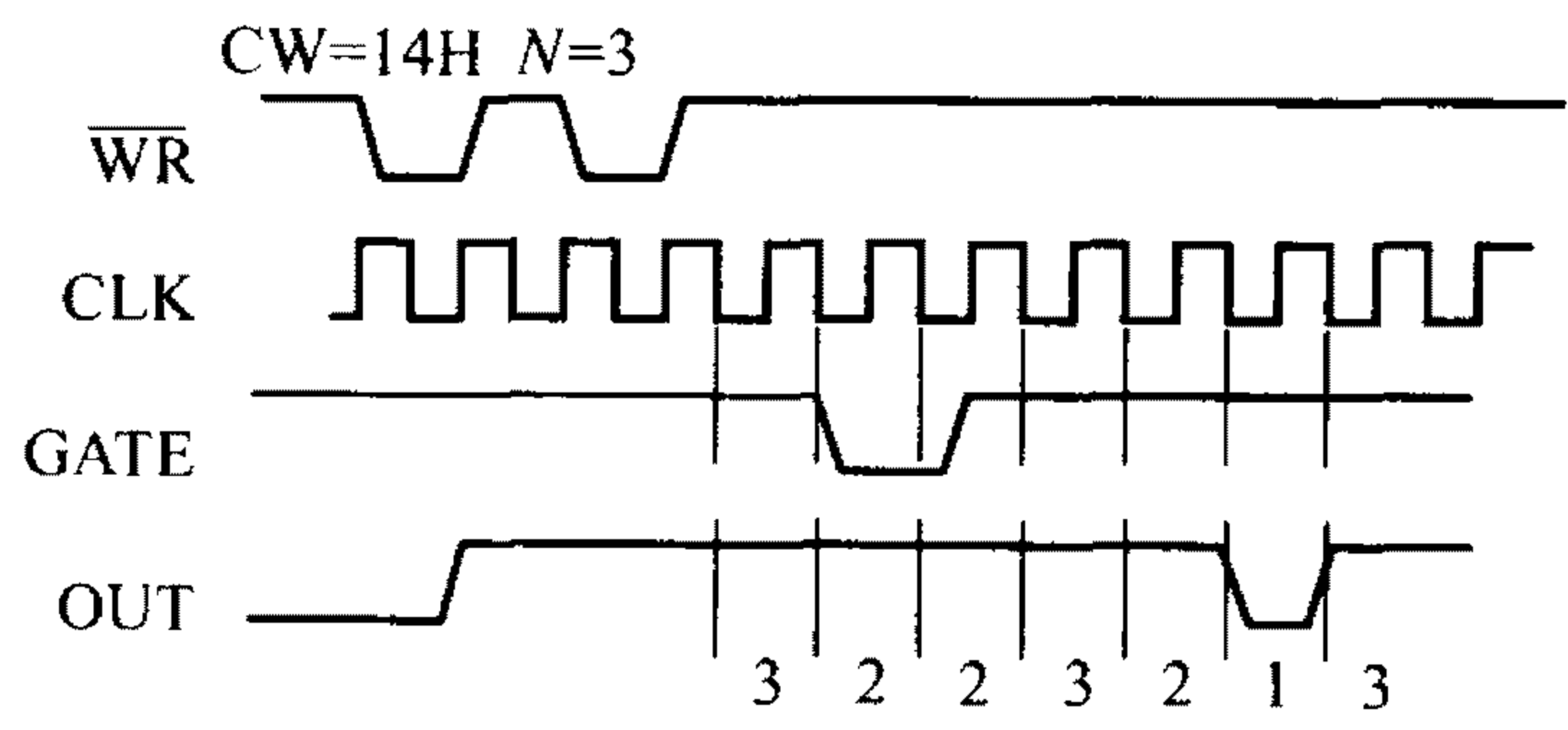


图 8-18 方式 2 时 GATE 信号的作用

- ③ 在计数过程中,如果 GATE 为高电平,写入新的计数初值时,不会影响正在进行的减 1 计数过程,只有计数器减到 1 之后,计数器才装入新的计数初值,并且按新的计数初值开始计数。如图 8-19 所示。

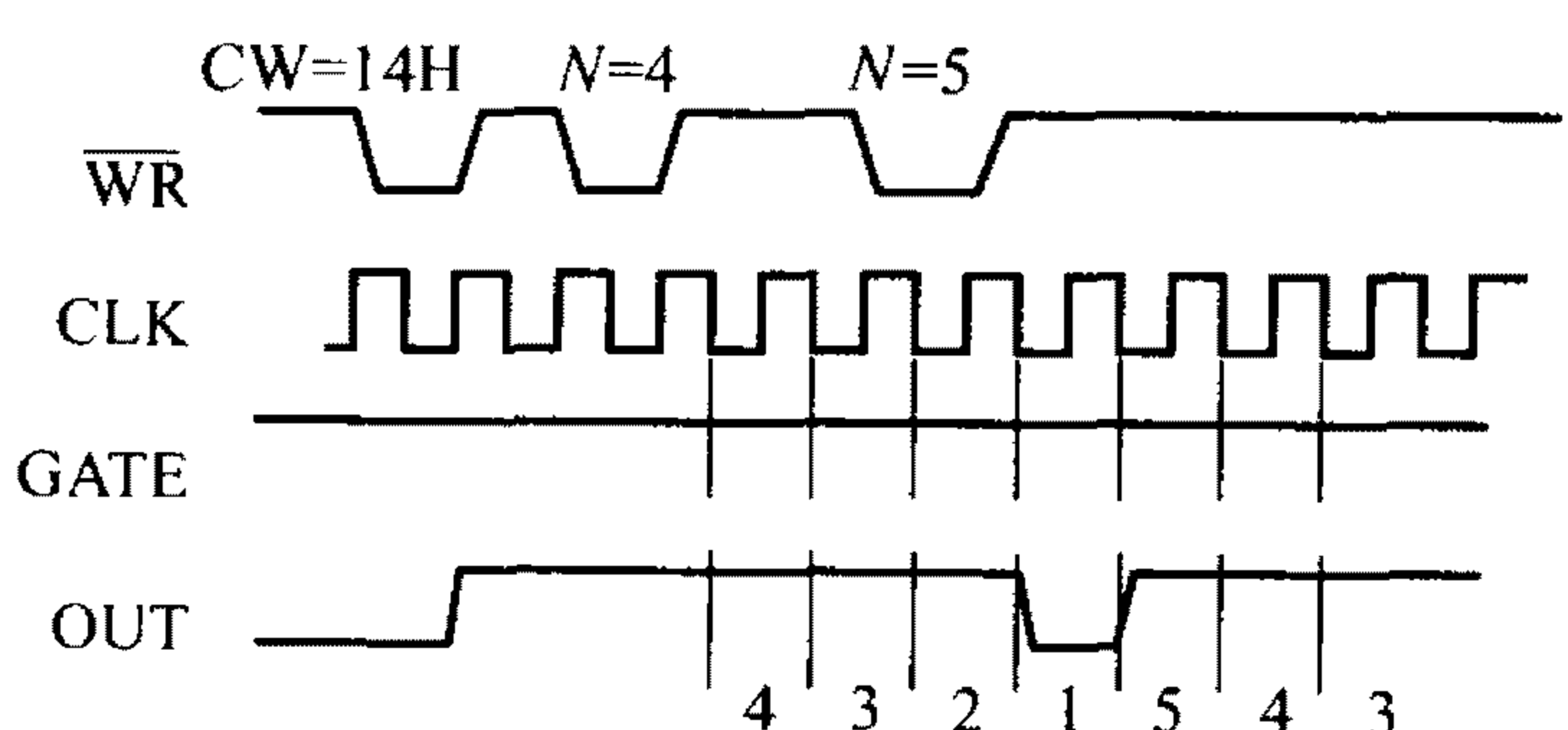


图 8-19 方式 2 在计数过程中改变计数初值

4. 方式3——方波发生器

方式3具有初值自动重装功能。

① 当计数初值为偶数时, 每来一个 CLK 脉冲, 计数值减 2, 当计数值减到 0 时输出端改变极性, 内部完成初值自动重装, 继续计数, 输出为 1:1 的方波, 正负脉冲的宽度均为 $N/2$ 个 CLK 周期, 计数过程如图 8-20 所示。

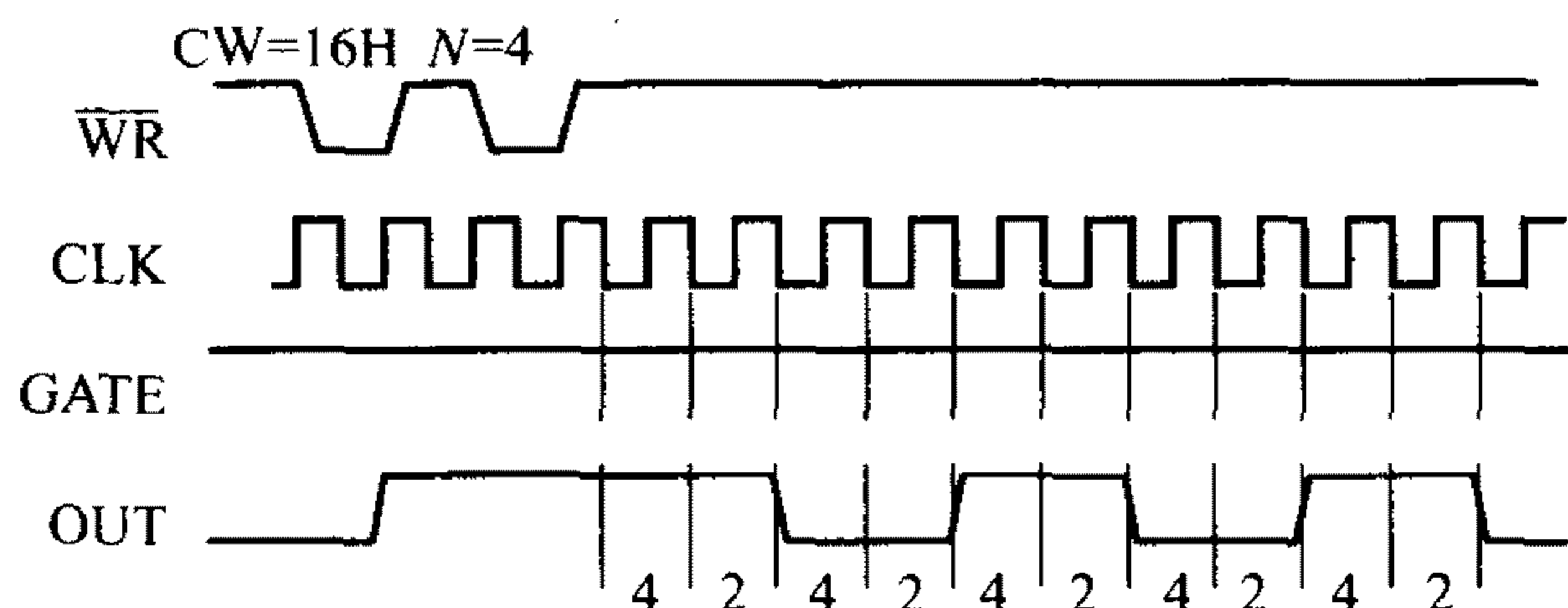


图 8-20 方式3计数值为偶数时的波形

② 如果计数初值为奇数, 计数过程如图 8-21 所示。

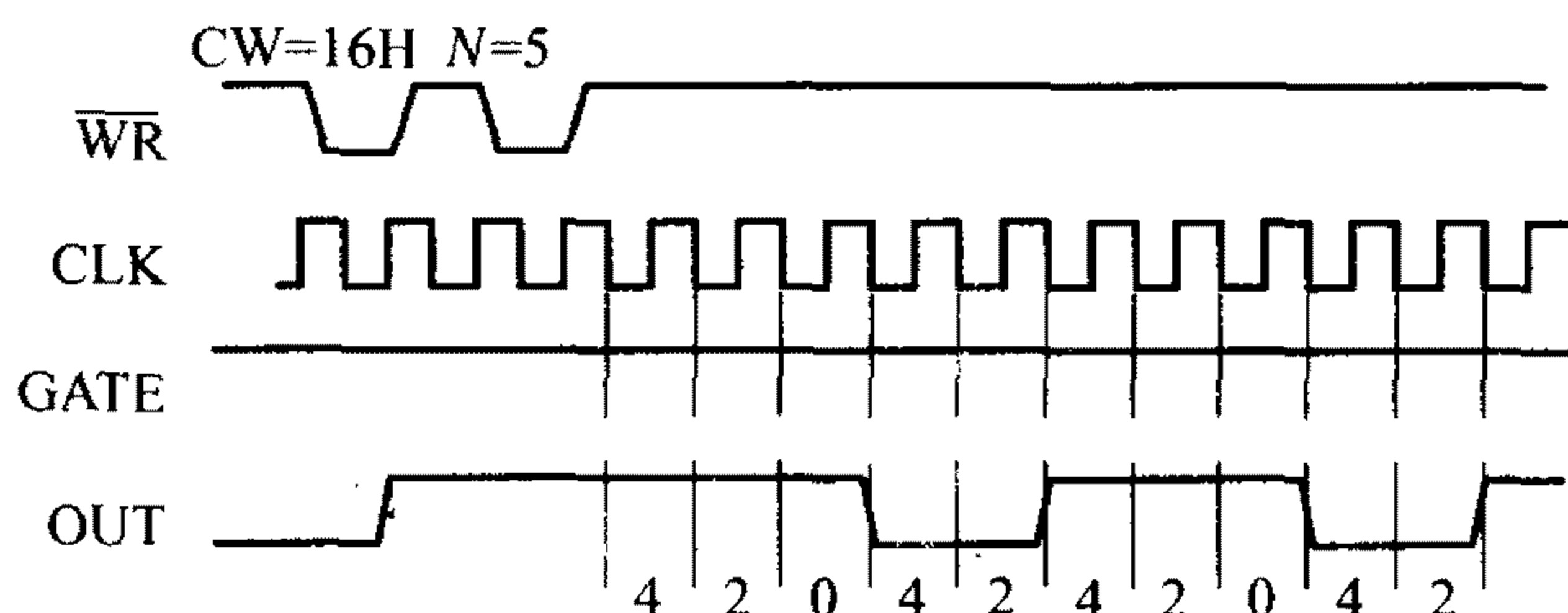


图 8-21 方式3计数值为奇数时的波形

在图 8-21 中, 输出正脉冲宽度 = $T_{CLK} \times (N+1)/2$; 输出负脉冲宽度 = $T_{CLK} \times (N-1)/2$ 。

实验证明: 实际装入的初值以及自动重装的初值, 均为编程时写入的初值减 1。

在输出正脉冲期间, 每一个 T_{CLK} 使计数值减 2, 当计数值减到 -2 时, 输出端变成低电平, 内部完成初值重装, 重装的初值为编程时写入的初值减 1。

在输出负脉冲期间, 每一个 T_{CLK} 使计数值减 2, 当计数值减到 0 时, 输出端变成高电平, 内部完成初值重装, 重装的初值为编程时写入的初值减 1。

5. 方式4——软件触发的单脉冲发生器

8254 工作在方式 4 时, 其工作波形如图 8-22 所示。

① 写入控制字后, OUT 输出高电平, 若当前 GATE 为高电平, 写入计数初值后, 开始作减 1 计数, 当计数值减到 0 时, OUT 变低, 在 OUT 端输出一个宽度为一个 CLK 周期的负脉冲。

② 当 $GATE=1$ 时, 允许计数; $GATE=0$ 时, 停止计数。如图 8-23 所示。

③ 在计数过程中, 如果改变计数值, 则按新的计数值重新开始计数。如图 8-24 所示。

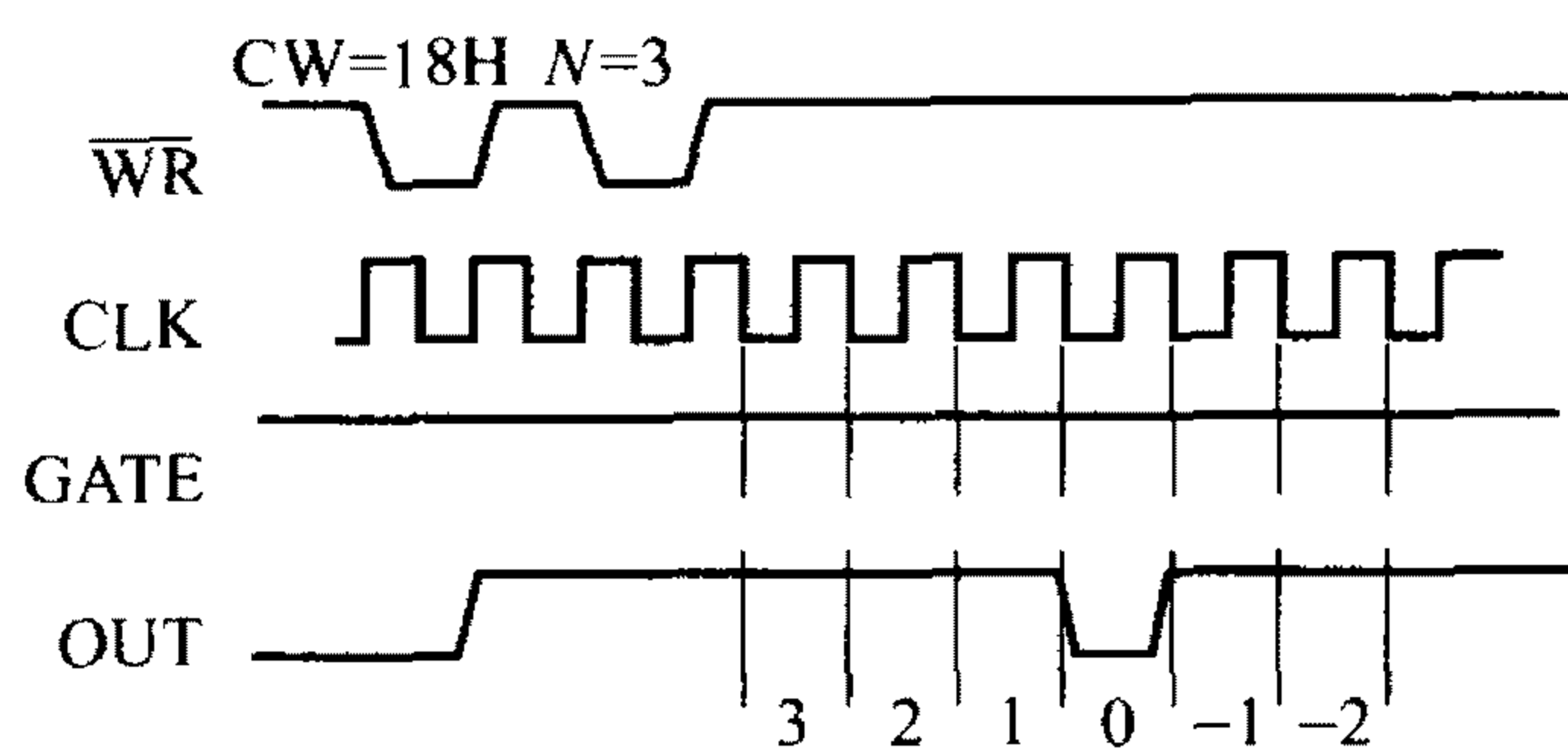


图 8-22 方式 4 波形图

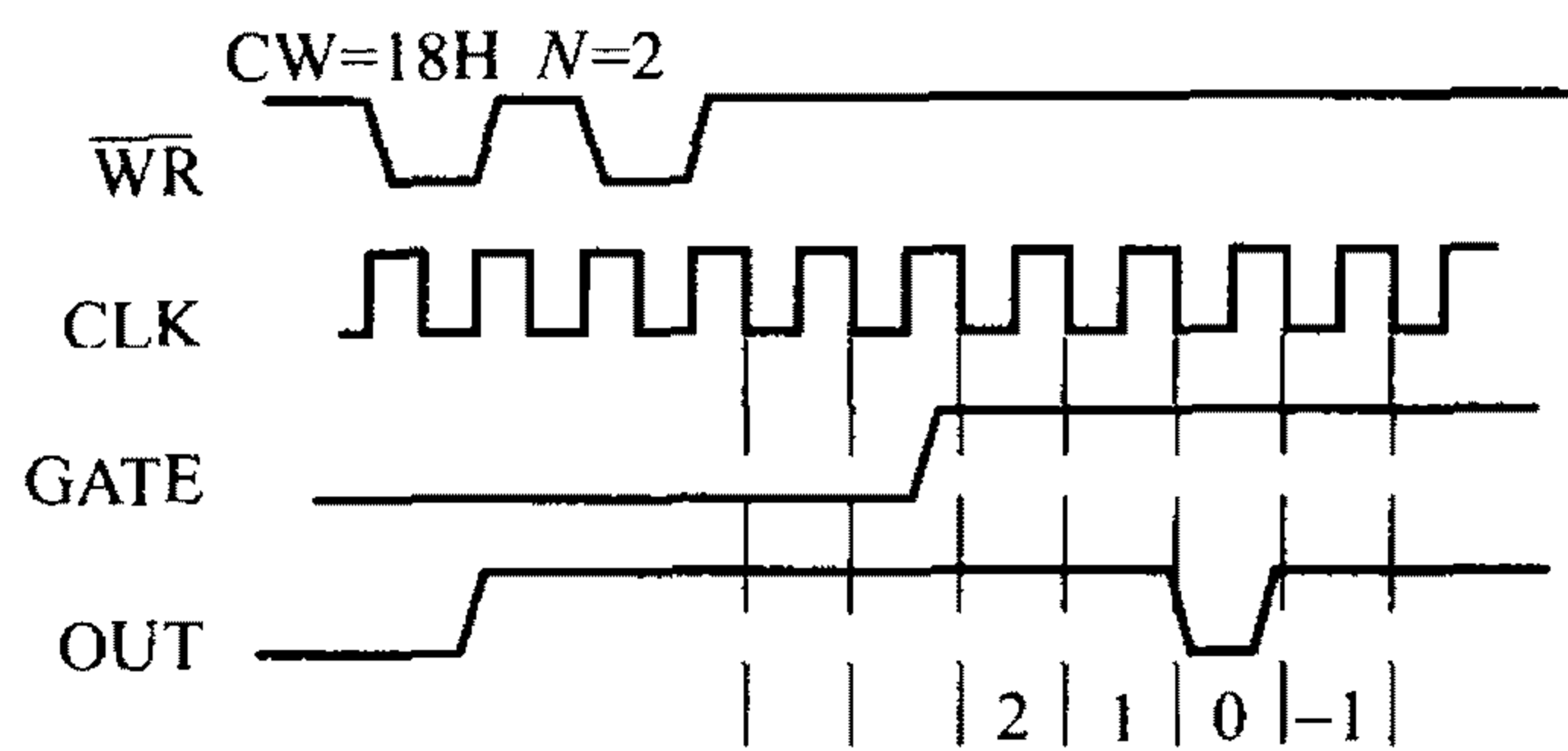


图 8-23 方式 4 时 GATE 信号的作用

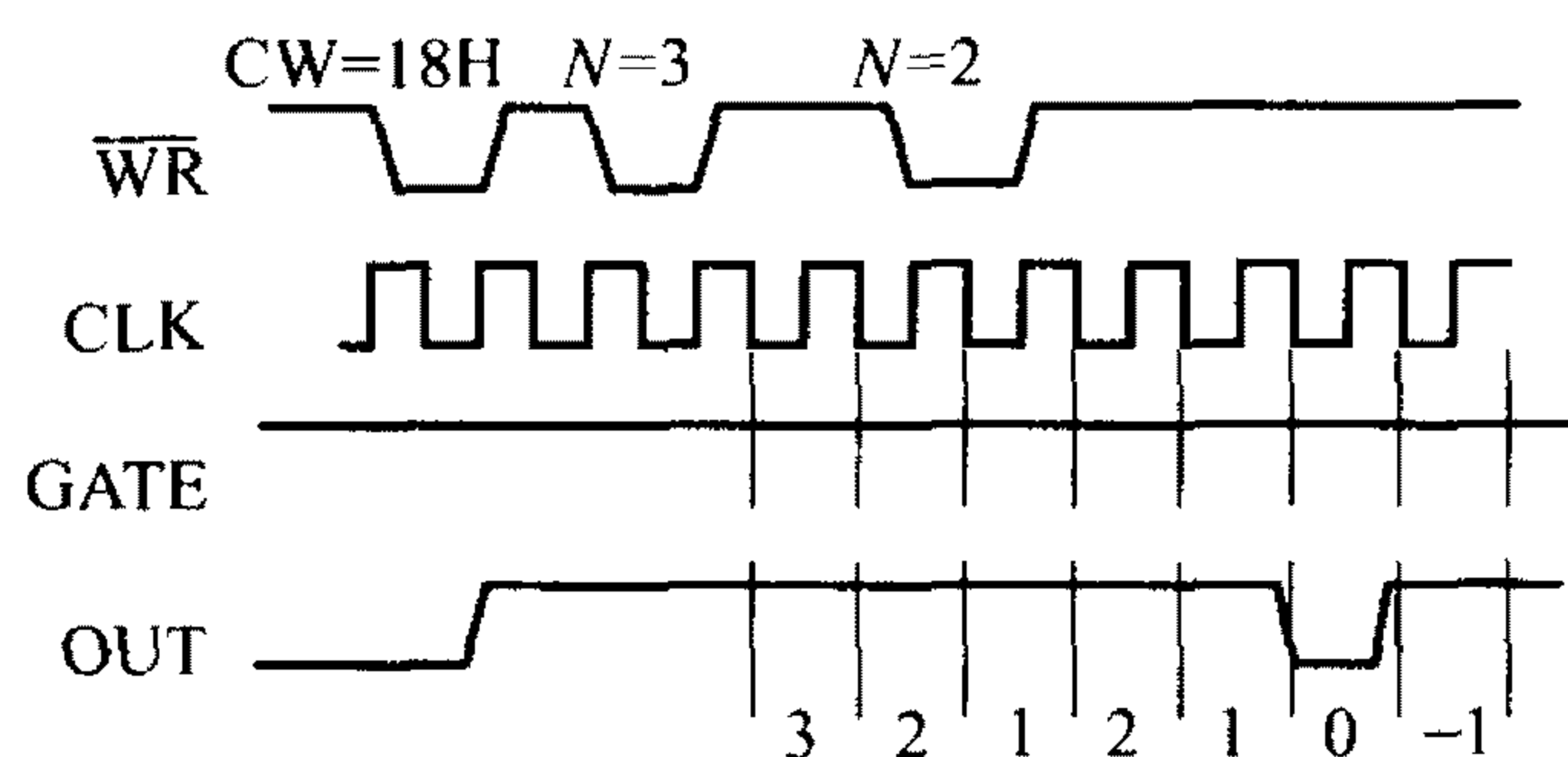


图 8-24 方式 4 在计数过程中改变计数初值

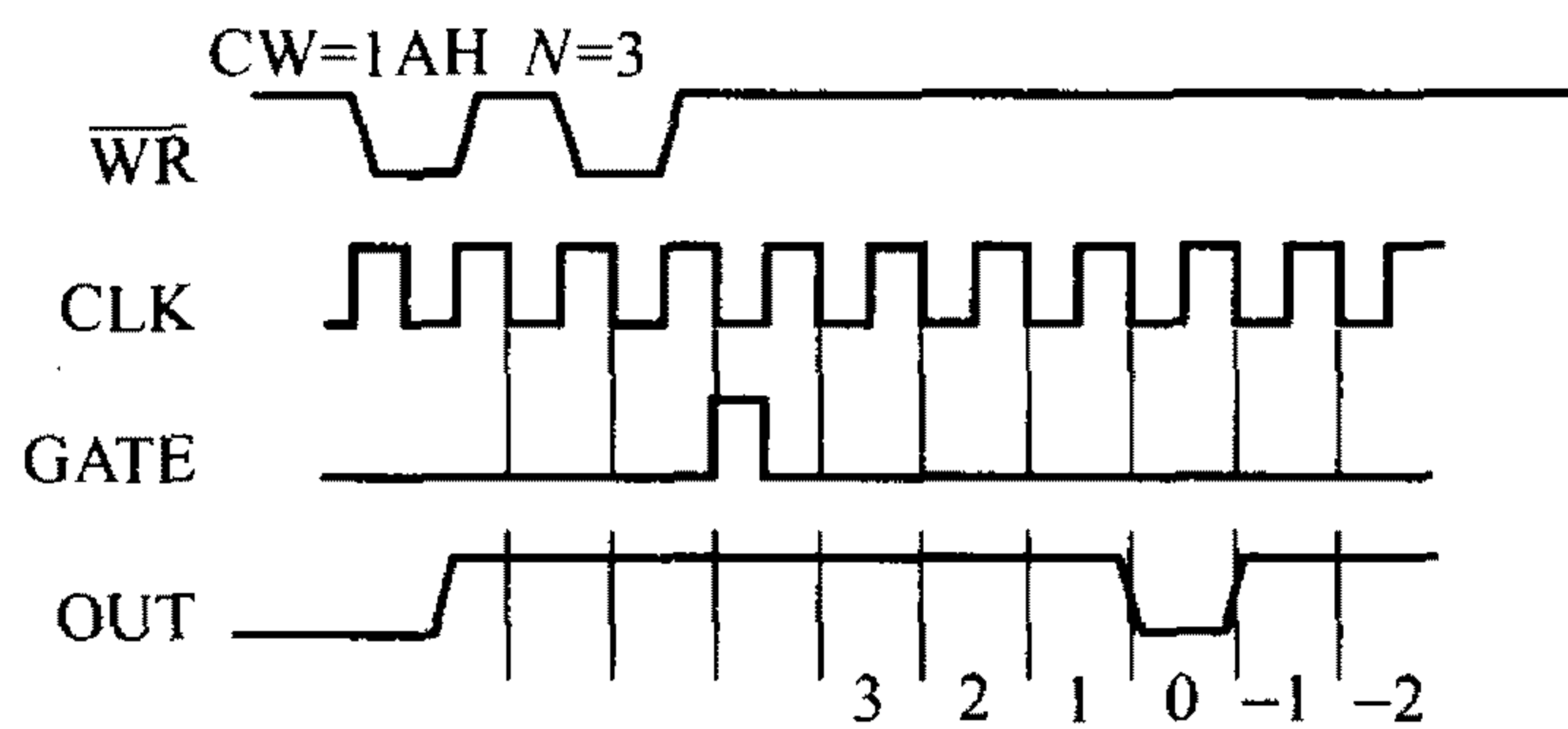


图 8-25 方式 5 波形图

6. 方式 5——硬件触发的单脉冲发生器

8254 工作在方式 5 时,其工作波形如图 8-25 所示。

① 写入控制字后,OUT 端输出高电平。写入计数初值后,只有在 GATE 端出现 0→1 跃变时,计数初值才能装入计数器,开始作减 1 计数,当计数值减为 0 时,OUT 端输出一个 CLK 周期的负脉冲。

② 在计数过程中,若 GATE 端再次出现 0→1 跃变,则计数初值重新装入计数器,作减 1 计数。如图 8-26 所示。

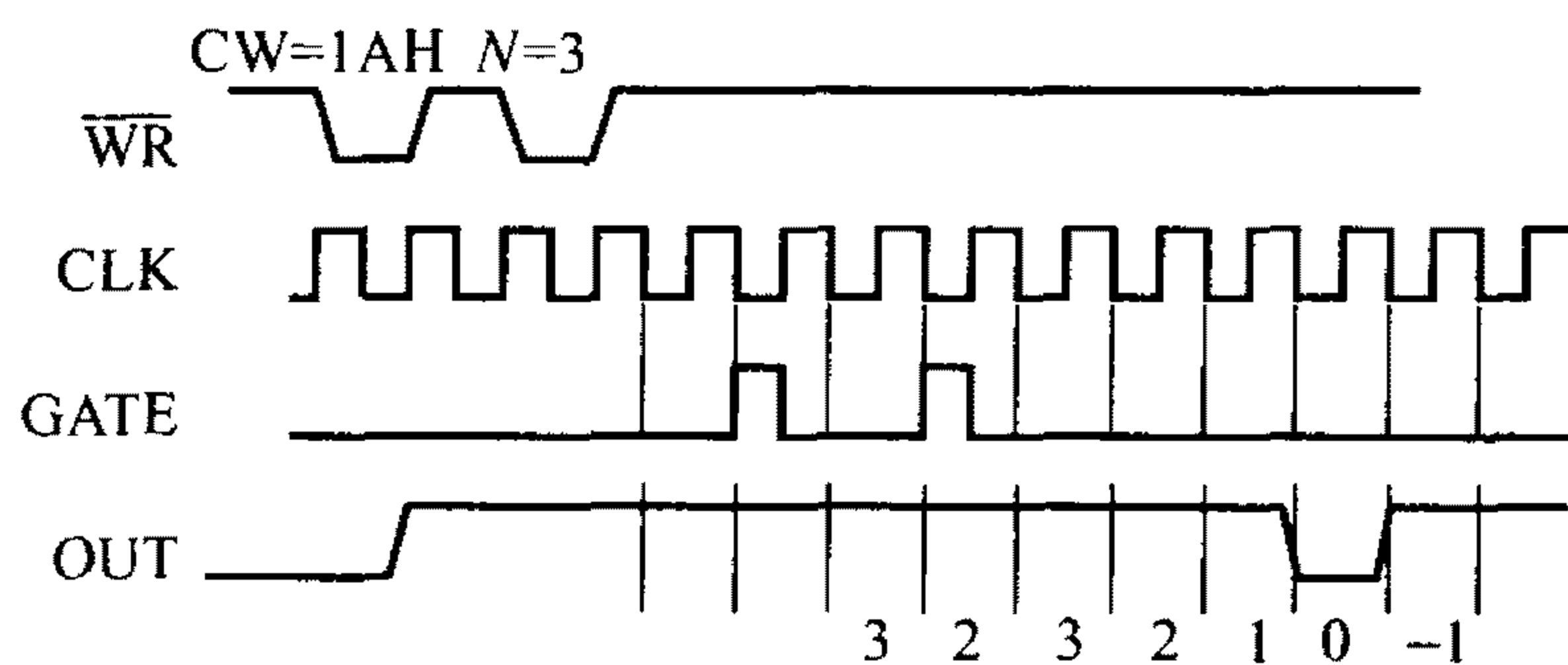


图 8-26 方式 5 时 GATE 信号的作用

③ 在计数过程中,如果改变计数初值,不影响当前计数过程;若有 GATE 上升沿触发,则按新的计数初值重新开始计数。如图 8-27 所示。

方式 5 与方式 1 的区别是:方式 5 输出的单脉冲(负)宽度为一个 CLK 周期,而方式 1 输出的单脉冲(负)宽度为 N 倍的 CLK 周期(N 为计数初值)。

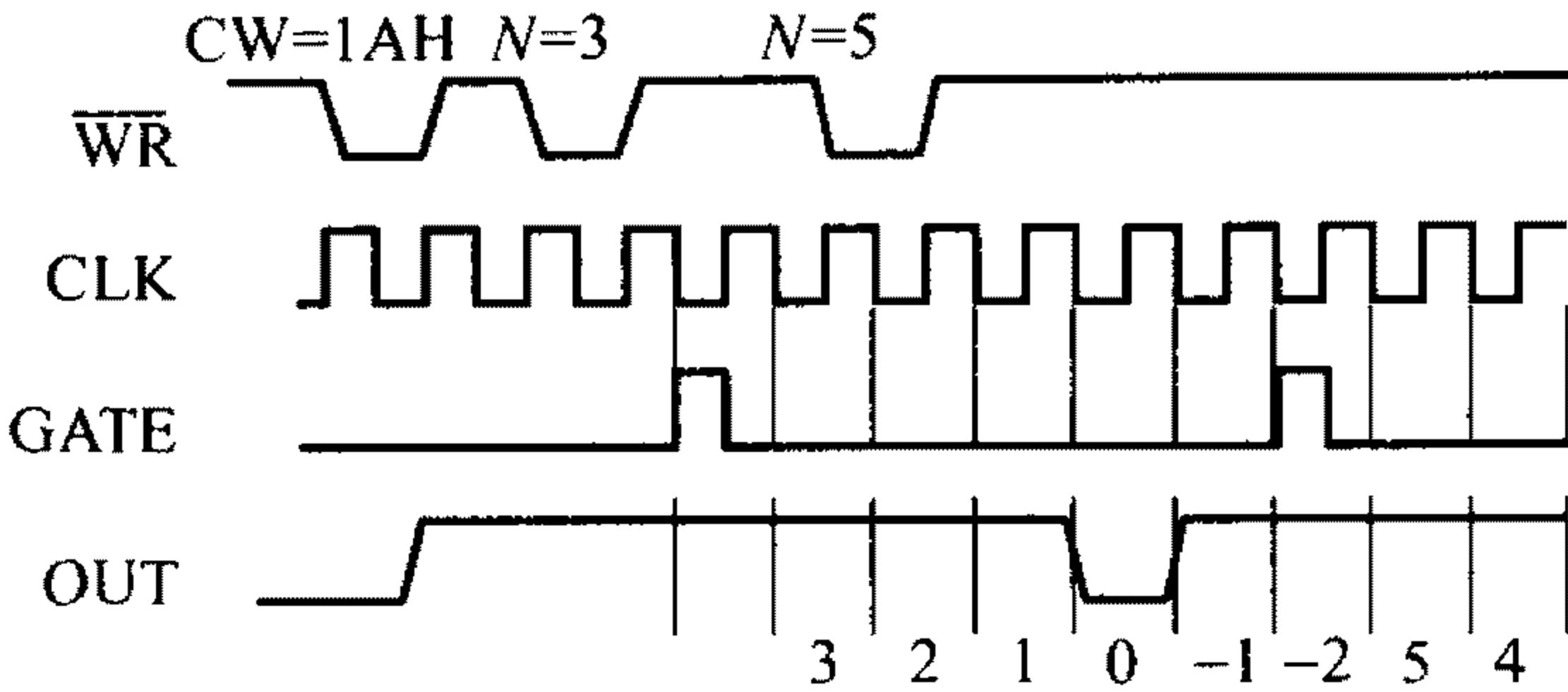


图 8-27 方式 5 在计数过程中改变计数初值

【小结】 表 8-2 列出了 8254 六种工作方式的比较。

表 8-2 8254 的 6 种工作方式比较(N 为计数初值)

		方式 0	方式 1	方式 2	方式 3	方式 4	方式 5
功能		计数结束输出正跃变信号	单脉冲发生器	频率发生器	方波发生器	单脉冲发生器	单脉冲发生器
启动方式		写入计数值(软件)启动	外部触发(硬件)启动	写入计数值(软件)启动	写入计数值(软件)启动	写入计数值(软件)启动	外部触发(硬件)启动
输出波形		写入计数值 N 后,经过 N + 1 个 CLK 输出为高	宽度为 N 个 CLK 周期的负脉冲	宽度为一个 CLK 周期的负脉冲	见备注(1)(2)	宽度为一个 CLK 周期的负脉冲	宽度为一个 CLK 周期的负脉冲
初值重数		—	—	初值自动重装	初值自动重装	—	—
计数过程中改变计数初值		立即有效	外部触发后有效	计数到 1 后有效	1. 外部触发有效 2. 计数结束后有效	立即有效	外部触发后有效
门控信号 GATE 的作用	GATE=0	停止计数	—	停止计数	停止计数	停止计数	—
	上升沿	—	启动计数	启动计数	启动计数	—	启动计数
	GATE=1	允许计数	—	允许计数	允许计数	允许计数	—

备注: (1) N 为偶数时, 正负脉宽均为 N/2 个 CLK 周期。
(2) N 为奇数时, 正脉宽为(N+1)/2 个 CLK 周期, 负脉宽为(N-1)/2 个 CLK 周期。

8.3.4 8254 的控制字与编程方法

1. 8254 的控制字/状态字

8254 的控制字有两个, 一个用来设置计数器的工作方式, 称为方式控制字; 另一个用来设置读出命令, 称为读出控制字。这两个控制字共用一个控制端口地址, 由标识位来区分。

(1) 方式控制字

方式控制字格式如图 8-28 所示。

- 计数器选择

D₇ D₆ = 00, 表示选择 0 号计数器。

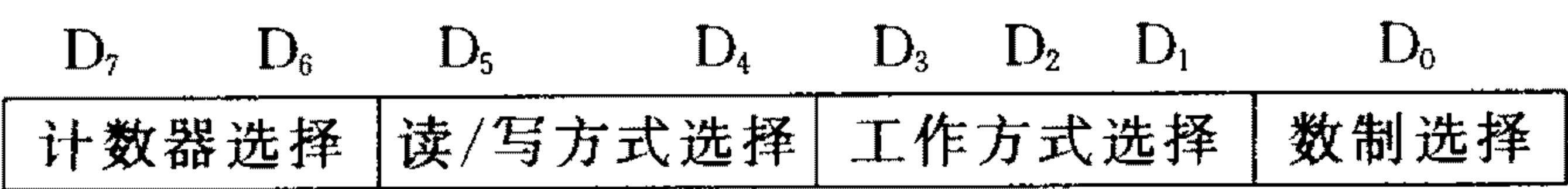


图 8-28 8254 控制字

D₇ D₆=01,表示选择 1 号计数器。
D₇ D₆=10,表示选择 2 号计数器。
D₇ D₆=11,读出控制字标志。

- 读/写方式选择
- D₅ D₄=00,表示锁存计数器的当前计数值,以便读出检查。
D₅ D₄=01,表示写入时,只写低 8 位计数值,高 8 位置 0;读出时,只能读出低 8 位计数值。
D₅ D₄=10,表示写入时,只写高 8 位计数值,低 8 位置 0;读出时,只能读出高 8 位计数值。
D₅ D₄=11,表示先读/写低 8 位计数值,后读/写高 8 位计数值。
- 工作方式选择
- D₃ D₂ D₁=000,计数器工作在方式 0。
D₃ D₂ D₁=001,计数器工作在方式 1。
D₃ D₂ D₁=×10,计数器工作在方式 2。
D₃ D₂ D₁=×11,计数器工作在方式 3。
D₃ D₂ D₁=100,计数器工作在方式 4。
D₃ D₂ D₁=101,计数器工作在方式 5。
- 数制选择
- 当 D₀=0 时,计数初值为二进制数,减 1 计数器按二进制规律减 1。初值范围是 0000H~FFFFH,其中 0000H 代表 65536。
当 D₀=1 时,计数初值为 BCD 码,减 1 计数器按十进制规律减 1。初值范围是 0000H~9999H,其中 0000H 代表十进制数 10000。

(2) 读出控制字

读出控制字的格式如图 8-29 所示。

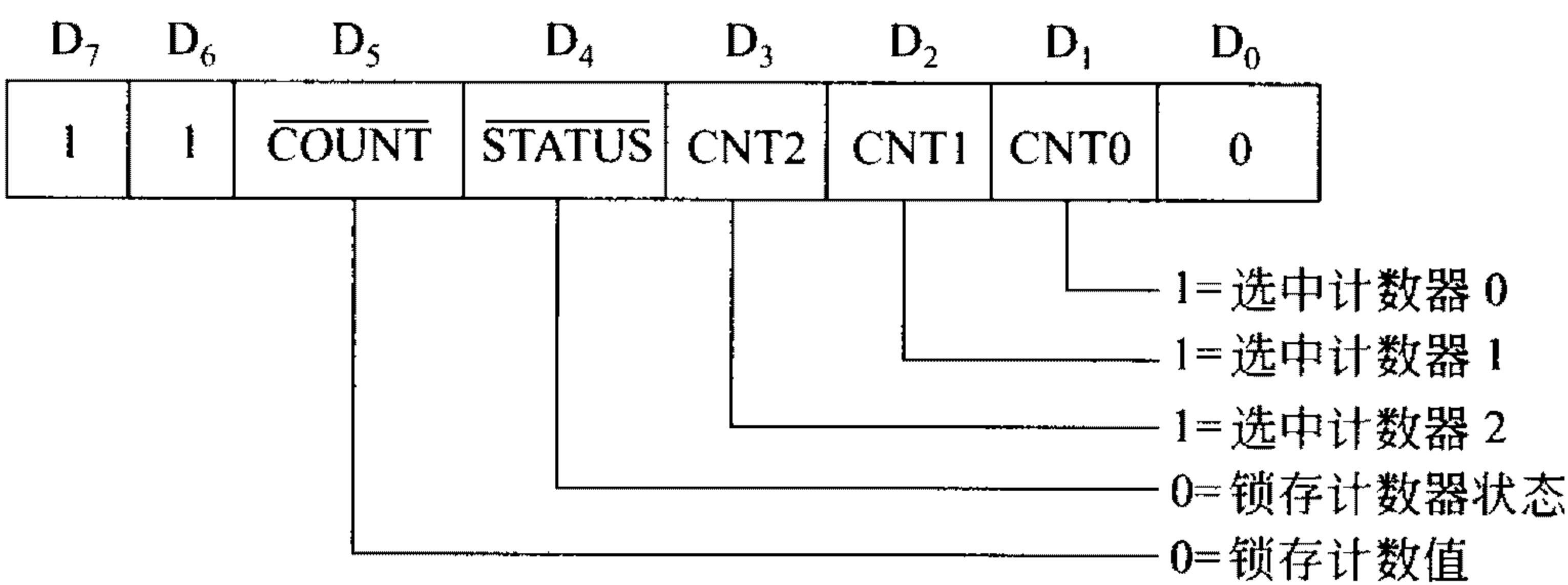


图 8-29 读出控制字格式

读出控制字 D₇ D₆ 必须为 11,D₀ 位必须为 0。D₅=0 锁存计数值,以便 CPU 读取;D₄=0 将状态信息锁存入状态寄存器;D₃~D₁ 为计数器选择,不论是锁存计数值还是锁存状态信息,都不影响计数。读出命令能同时锁存几个计数器的计数值/状态信息,当

CPU 读取某一计数器的计数值/状态信息时,该计数器自动解锁,但其他计数器不受影响。

(3) 状态字

状态字格式如图 8-30 所示。

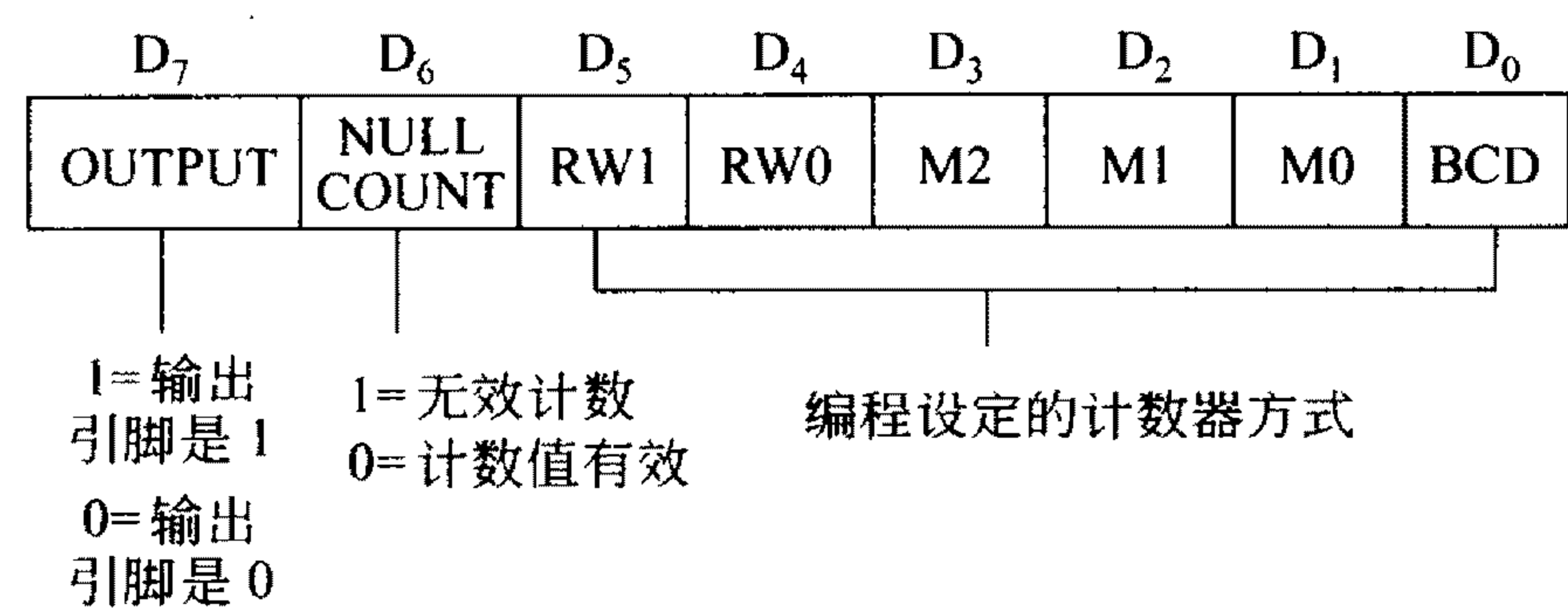


图 8-30 状态字格式

D₅~D₀ 意义与方式控制字的对应位意义相同,D₇ 表示 OUT 引脚的输出状态,D₇ 位=1,表示 OUT 引脚为高电平,D₇ 位=0,表示 OUT 引脚为低电平。D₆ 表示计数初值是否已装入减 1 计数器,D₆=0,表示已装入,可以读取计数器。

2. 8254 初始化编程

8254 是可编程芯片,使用之前需要进行初始化编程。初始化编程分两步进行:首先向控制字寄存器写入方式控制字,对使用的计数器规定其工作方式等;然后向使用的计数器写入计数初值。

【例 8.3.1】 设 8254 端口地址为 40H~43H,要求 2 号计数器工作在方式 1,按 BCD 码计数,计数初值为十进制数 4000。试写出初始化程序段。

解 根据题意,设定按 BCD 码计数,计数初值为十进制数 4000,所以编程时写入的计数初值应为 4000H。由于计数初值低 8 位为 0,控制字可设定读/写操作为只写高 8 位,低 8 位由 8254 自动置 0,所以控制字为 A3H。

程序段如下:

```
MOV    AL,0A3H
OUT    43H,AL      ;写控制字
MOV    AL,40H
OUT    42H,AL      ;写初值为 4000
```

3. 读取当前计数值

8254 任一计数器的计数值,可用输入指令读取。由于计数器为 16 位,因而要分两次读。

读操作有以下 5 种形式,当初始化编程规定的读/写方式为先低 8 位后高 8 位时:

- ① 使 GATE=0,停止计数,然后对相应的计数器端口进行两次读操作,第一次读出的是低 8 位计数值,第二次读出的是高 8 位计数值。这种方法在系统机中无法实现。
- ② 在计数过程中,先向 8254 控制寄存器写入一个 D₇ D₆=计数器编号,D₅ D₄=00 的控制字,锁存相应计数器的当前计数值,然后再对相应的计数器端口进行两次读操作,

依次读出计数值的低 8 位和高 8 位。

③ 在计数过程中,向 8254 控制寄存器写入读出命令,分 3 种情况:

如果读出命令仅锁存相应计数器的状态信息,则对相应计数器端口进行一次读操作,即可读出状态信息。

如果读出命令仅锁存相应计数器的计数值,则对相应计数器端口进行两次读操作,依次读出计数值的低 8 位和高 8 位。

如果读出命令同时锁存计数器的计数值和状态信息,则要对相应计数器端口执行三次读操作,第一次读出的是状态信息,第二次读出的是当前计数值的低 8 位,第三次读出的是当前计数值的高 8 位。

【例 8.3.2】 设 8254 端口地址为 40H~43H,试写出程序段,读取 2 号计数器的当前计数值。

程序段如下:

```
MOV    AL,84H    ;计数器 2 号的锁存命令
OUT    43H,AL    ;写入控制寄存器
IN     AL,42H    ;读低 8 位
MOV    CL,AL     ;存于 CL 中
IN     AL,42H    ;读高 8 位
MOV    CH,AL     ;存于 CH 中
```

8.3.5 8254 在微型计算机系统中的应用

在 PC 系列机中,8254 是 CPU 外围支持电路之一,提供动态存储器刷新定时、系统时钟中断及发声系统音调控制等功能。8254 的初始化是在系统启动时由 BIOS 完成的。

图 8-31 是 8254 在 IBM PC/AT 中的应用示意图。

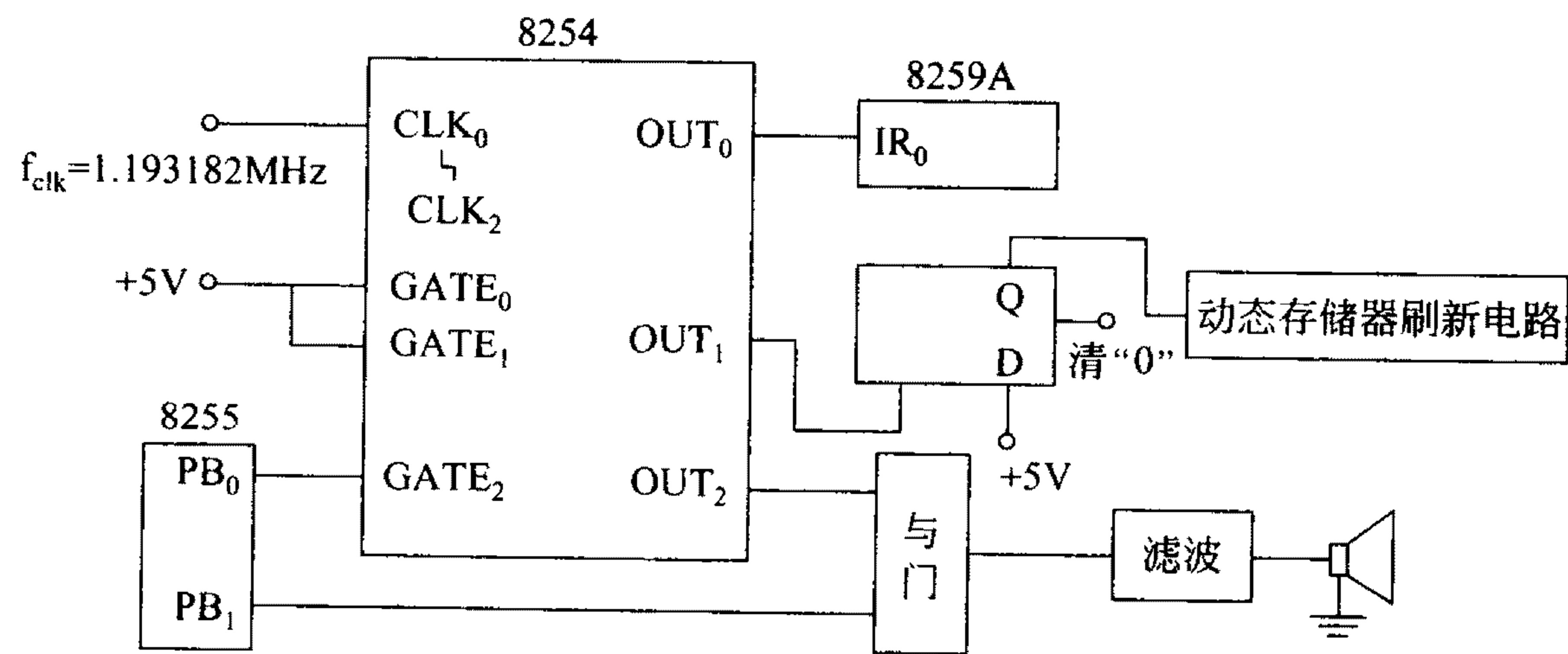


图 8-31 8254 在 IBM PC/AT 中的应用示意图

表 8-3 给出了 8254 在 PC/AT 中的使用情况。

系统机中 8254 的端口地址为 40H~43H,8255 B 口的地址为 61H。

ROM-BIOS 中的 3 个计数器初始化程序段如下:

表 8-3 8254 在 PC/AT 中的使用现状

计数器	工作方式	计数方式	初 值	控制字	Tout	fout
0 #	3	二进制	0	36H	55ms	
1 #	2	二进制	12H	54H	15.1μs	66.827KHz
2 #	3	二进制	533H	B6H		约 900Hz

• 计数器 0 用于定时(约 55ms)中断

```
MOV    AL,00110110B    ;方式 3,二进制计数
OUT    43H,AL
MOV    AL,0             ;初值为 0000H
OUT    40H,AL
OUT    40H,AL
```

• 计数器 1 用于动态存储器刷新定时(每隔 15μs 提出一次请求)

```
MOV    AL,01010100B    ;方式 2,只写低 8 位,二进制计数
OUT    43H,AL
MOV    AL,12H           ;初值为 18
OUT    41H,AL
```

• 计数器 2 用于产生约 900Hz 的方波送至扬声器

```
MOV    AL,10110110B    ;方式 3,二进制计数
OUT    43H,AL
MOV    AX,0533H         ;初值为 533H
OUT    42H,AL           ;先写低 8 位
MOV    AL,AH
OUT    42H,AL           ;再写高 8 位
```

【例 8.3.3】 编制程序,使 PC 系列机 8254 的计数器产生 800Hz 的方波,经滤波后送至扬声器发声,当键盘有任意按键时声音停止。

解 ① 使 8255 PB₀、PB₁=1,接通扬声器。

② 利用 8254 计数器 2 号产生 800Hz 方波。

计数初值

$$N=f_{CLK_2} \div f_{OUT_2}=1.193182MHz \div 800Hz$$

程序如下:

```
CODE    SEGMENT
        ASSUME CS:CODE
BEG:    IN      AL,61H      ;读 PB 口当前状态
        OR      AL,03H     ;使 PB0,PB1 均为 1
        OUT     61H,AL     ;写新的 PB 口值,接通扬声器
        MOV     DX,12H
        MOV     AX,34DEH   ;1.193182×106=1234DEH
        MOV     CX,800
        DIV     CX         ;计数初值→AX
```

```
OUT    42H,AL      ;先写低 8 位
MOV    AL,AH
OUT    42H,AL      ;再写高 8 位,OUT2 输出 800Hz 方波
MOV    AH,1
INT    21H         ;等待按键
IN     AL,61H      ;读 PB 口当前状态
AND    AL,0FCH     ;使 PB1、PB0 均为 0
OUT    61H,AL      ;扬声器停止工作
MOV    AH,4CH
INT    21H
CODE   ENDS
END    BEG
```

8.4 发声系统与音乐程序设计

8.4.1 PC 系列机发声系统

从图 8-31 看出,发声系统以定时器 2 号计数器为核心。系统初始化后,2 号计数器工作在“方波发生器”方式;计数初值(533H)为二进制数;初值的写入顺序为先低 8 位,后高 8 位。CLK₂ 输入频率为 1.193182MHz。计数初值 533H,使 OUT₂ 输出的方波大约为 900Hz。改变计数初值就能得到不同频率的方波输出。OUT₂ 端输出的方波,经过简单的滤波,驱动机箱内的喇叭。

1. 发声控制

发声系统受 8255 芯片 B 口的两个输出端线(PB₀、PB₁)的控制。PB₀ 为“1”,使 GATE₂ 为 1,2 号计数器能正常计数。PB₁ 为“1”,打开输出控制门。因此执行下面的 OPEN 子程序可以打开扬声器,执行 CLOSE 子程序则关闭扬声器而不影响PB₇~PB₂。

OPEN	PROC		CLOSE	PROC	
	PUSH	AX		PUSH	AX
	IN	AL,61H		IN	AL,61H
	OR	AL,00000011B		AND	AL,11111100B
	OUT	61H,AL		OUT	61H,AL
	POP	AX		POP	AX
	RET			RET	
OPEN	ENDP		CLOSE	ENDP	

2. 演奏单音符

一首乐曲由若干音符组成,一个音符对应一个频率。将一个频率对应的计数初值写入 2 号计数器,即能发出规定的音调,计数初值的计算公式如下:

$$\text{计数初值} = 1193182 \div \text{输出频率}$$

1.193182MHz 转换成 16 进制数应为 1234DEH,因此,在接通扬声器的前提下,执行

下列程序段即可发出与输出频率对应的音调。

```

...      ...
MOV     DX,12H
MOV     AX,34DEH
MOV     CX,频率值
DIV     CX
OUT     42H,AL
MOV     AL,AH
OUT     42H,AL
...      ...

```

如何控制每一个音符的演奏时间呢？最简便的方法是调用 INT 15H 的 86H 号子程序。

但是，实验证明这一延时功能在 Windows 2000(含 2000)以上的命令行方式下不能起到延时的作用，参照例 5.10.9，改用 INT 21H 的 2CH,2DH 功能可以设计出比较精确的延时程序。

3. 音符频率表

表 8-4 给出了各种音符的频率：

表 8-4 音符频率表

单位：Hz

音符 音调	1	2	3	4	5	6	7
A	221	248	278	294	330	371	416
B	248	278	312	330	371	416	467
C	131	147	165	175	196	221	248
D	147	165	185	196	221	248	278
E	165	185	208	221	248	278	312
F	175	196	221	234	262	294	330
G	196	221	248	262	294	330	371
音符 音调	1	2	3	4	5	6	7
A	441	495	556	589	661	742	833
B	495	556	624	661	742	833	935
C	262	294	330	350	393	441	495
D	294	330	371	393	441	495	556
E	330	371	416	441	495	556	624
F	350	393	441	467	525	589	661
G	393	441	495	525	589	661	742
音符 音调	1	2	3	4	5	6	7
A	882	990	1112	1178	1322	1484	1665
B	990	1112	1248	1322	1484	1665	1869
C	525	589	661	700	786	882	990

续表

音符 音调	$\dot{1}$	$\dot{2}$	$\dot{3}$	$\dot{4}$	$\dot{5}$	$\dot{6}$	$\dot{7}$
D	598	661	742	786	882	990	1112
E	661	742	833	882	990	1112	1248
F	700	786	882	935	1049	1178	1322
G	786	882	990	1049	1178	1322	1484

8.4.2 音乐程序设计举例

【例 8.4.1】 设计音乐程序,重复演唱“友谊地久天长”,直到按下任意键时停唱。歌谱如图 8-32 所示。

1=F 2/4

友谊地久天长

苏格兰民歌

5 | 1 . 1 1 3 | 2 . 1 2 3 | 1 . 1 3 5 | 6 . 6 6 | 5 . 3

1. 怎 能 忘 记 旧 日 朋 友, 心 中 能 不 怀 想, 旧 日 朋

2. 我 们 曾 经 终 日 游 荡, 在 故 乡 的 青 山 上, 我 们 也

3. 我 们 也 曾 终 日 逍 遥, 荡 桨 在 绿 波 上, 但 如 今

4. 我 们 往 日 情 意 相 投, 让 我 们 紧 握 手, 让 我 们

3 1 | 2 . 1 2 3 | 1 . 6 6 5 | 1 . 6 | 5 . 3 3 1 | 2 . 1 2 . 6 |

友 岂 能 相 忘, 友 谊 地 久 天 长:

曾 历 尽 苦 辛, 到 处 奔 波 流 浪:

却 劳 燕 分 飞, 远 隔 大 海 重 洋:

来 举 杯 畅 饮, 友 谊 地 久 天 长:

5 . 3 3 5 | 6 . 1 | 5 . 3 3 1 | 2 . 1 2 3 | 1 . 6 6 5 | 1 . :

谊 万 岁! 举 杯 痛 饮, 同 声 歌 颂 友 谊 地 久 天 长。

图 8-32 歌谱

【程序框图】(图 8-33 所示)

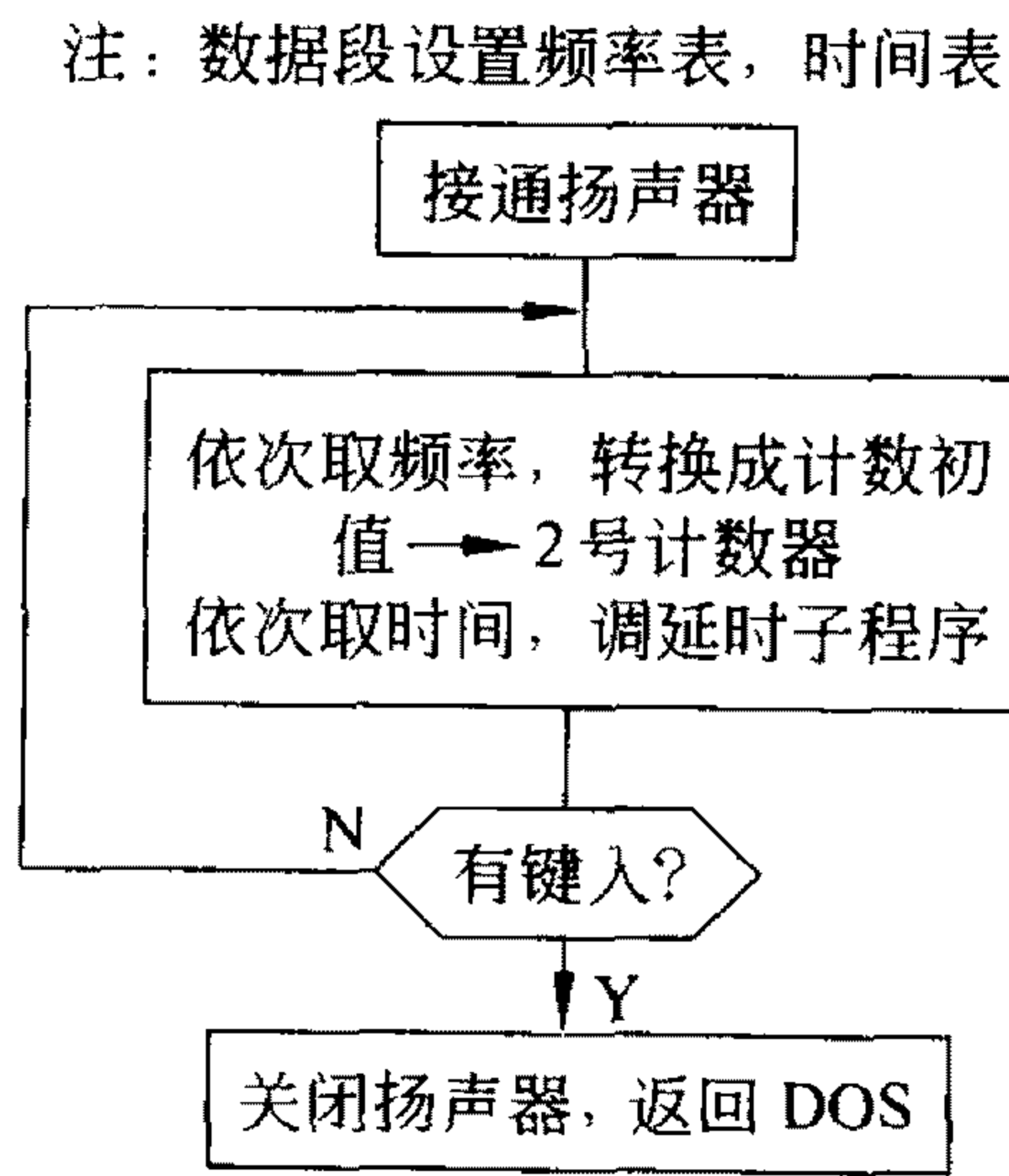


图 8-33 音乐程序框图



【设计思路】

① 首先在数据段设计频率表和时间表。将音符对应的频率值依次写入频率表中,各音符的演唱时间依次写入时间表。频率表和时间表的表项要一一对应,不能错位。频率表最后一项为 0 值,做为重复演唱的标志。

歌曲中的“休止符”如何处理? 遇到休止符不发声,理论上可以关闭扬声器实现,但在编程时这样处理比较麻烦,你可以让计数器产生一个很高的频率,例如“-1”,-1 经过汇编后就是 65535Hz,这个频率人耳已经听不到了。

② 时间表的内容有两种设置方法。

一种是在时间表中设置每一音符演唱的绝对时间。这种方法程序直观,但是开发速度慢,程序调试比较困难,特别是一首不熟悉的歌曲,对那些缺少音乐细胞的人来说,很难把握演唱的节奏,初期很难确定每个音符的演唱时间,因此调试十分困难。

另一种是在时间表中设置每个音符演唱的“单位时间”。

$$\text{单位时间} \times N = \text{演唱时间}$$

其中 N 为调试参数,用 EQU 伪指令定义,初值先行估算,调试时再修改。

如何确定音符演唱的单位时间呢?

我们知道,音符的演唱时间分为:1 拍、1/2 拍、1/4 拍、1/8 拍等。如果一首歌曲中,音符演唱的最短时间为 1/8 拍,则假定 1 拍的单位时间为 8,半拍的单位时间即为 4,1/4 拍的单位时间即为 2……

如果时间表中设置单位时间,只要对程序稍加修改,就可以使每一遍演唱的速度不同,极有趣味性。

此外,对于连续演唱的相同音符,其频率值可做适当修改,以达到最佳效果。

本程序的时间表,设置音符演唱的单位时间。

【程序清单】

```

;FILENAME: MUSIC.ASM
.486
DATA    SEGMENT USE16
TABF    DW      -1,262,350,352,350,441,393,350,393,441
        DW      350,352,441,525,589,588,589,525,441
        DW      440,350,393,350,393,441,350,293,294,262
        DW      350,589,525,441,440,350,393,350,393,589
        DW      525,441,440,525,589,700,525,441,440,350
        DW      393,350,393,441,350,294,292,262,350,0
TABT    DB      4,4,6,2,4,4,6,2,4,4
        DB      6,2,4,4,12,1,3,6,2
        DB      4,4,6,2,4,4,6,2,4,4
        DB      12,4,6,2,4,4,6,2,4,4
        DB      6,2,4,4,12,4,6,2,4,4
        DB      6,2,4,4,6,2,4,4,12
N        EQU      15
TTT      DW      0

```

```

DATA    ENDS
CODE    SEGMENT USE16
        ASSUME CS: CODE, DS: DATA
BEG:     MOV     AX, DATA
        MOV     DS, AX
OPEN:    IN      AL, 61H
        OR      AL, 00000011B
        OUT     61H, AL          ;接通扬声器
AGA:     MOV     SI, OFFSET TABF ;SI 是频率表指针
        MOV     DI, OFFSET TABT ;DI 是时间表指针
LAST:    CMP     WORD PTR [SI], 0 ;唱完一遍?
        JE      AGA             ;是, 转移
        MOV     DX, 12H
        MOV     AX, 34DEH
        DIV     WORD PTR [SI]   ;频率转换成计数初值
        OUT     42H, AL         ;低 8 位送 2 号计数器
        MOV     AL, AH
        OUT     42H, AL         ;高 8 位送 2 号计数器
        CALL    DELAY           ;延时
        ADD     SI, 2           ;频率表指针加 2
        INC     DI              ;时间表指针加 1
        MOV     AH, 1
        INT     16H             ;有键入?
        JZ      LAST           ;否
CLOSE:   IN      AL, 61H
        AND     AL, 11111100B
        OUT     61H, AL         ;关闭扬声器
        MOV     AH, 4CH
        INT     21H
;-----
DELAY    PROC                                ;延时子程序
        MOV     AL, N
        MUL     BYTE PTR [DI]
        MOV     TTT, AX          ;TTT=演唱时间
        MOV     AH, 2DH
        MOV     CX, 0
        MOV     DX, 0
        INT     21H
READ:    MOV     AH, 2CH
        INT     21H
        MOV     AL, 100
        MUL     DH
        MOV     DH, 0
        ADD     AX, DX           ;AX=延时时间(百分秒)

```

	CMP	AX, TTT
	JC	READ
	RET	
DELAY	ENDP	
CODE	ENDS	
	END	BEG

习 题

1. 外设为什么要通过接口电路和主机系统相连?
2. 接口电路的作用是什么? I/O 接口应具备哪些功能?
3. 什么是端口? 端口有几类?
4. I/O 端口有哪两种编址方式? PC 系列机中采用哪种编址方式?
5. 微型计算机系统和输入/输出设备交换信息的方式有几种? 这些方式各有什么特点?
6. 定时/计数器各通道的 CLK、GATE 信号各有什么作用?
7. 定时/计数器的 3 个通道在 PC 系列机中是如何应用的?
8. 说明定时/计数器 8254 的 GATE 信号在 6 种工作方式下的作用以及与时钟信号 CLK 的关系。
9. 简述定时/计数器 8254 方式 1、方式 4、方式 5 的工作特点。
10. 系统机定时/计数器的通道 0 定时周期最长是多少? 要实现长时间定时, 应采取什么措施? 如果采用外扩 8254 定时/计数器实现长时间定时, 应采取哪些措施?

中 断 系 统

以 80386/80486 为核心的微型计算机系统工作在实地址模式和保护模式下,其中断机制有很大区别,本章仅介绍实地址模式下的中断原理。

9.1 中断的基本概念

1. 中断概念的引入

当我们用查询方式从输入设备的接口电路中读取数据时,先要查询接口电路的状态,只有当接口电路准备好一个数据的时候才能读取,否则 CPU 只能等待。

能否使得“数据输入”与“CPU 执行现行程序”二者“同步”进行呢? 由此引入了“中断”的概念。图 9-1 是运用中断技术完成数据输入的示意图。

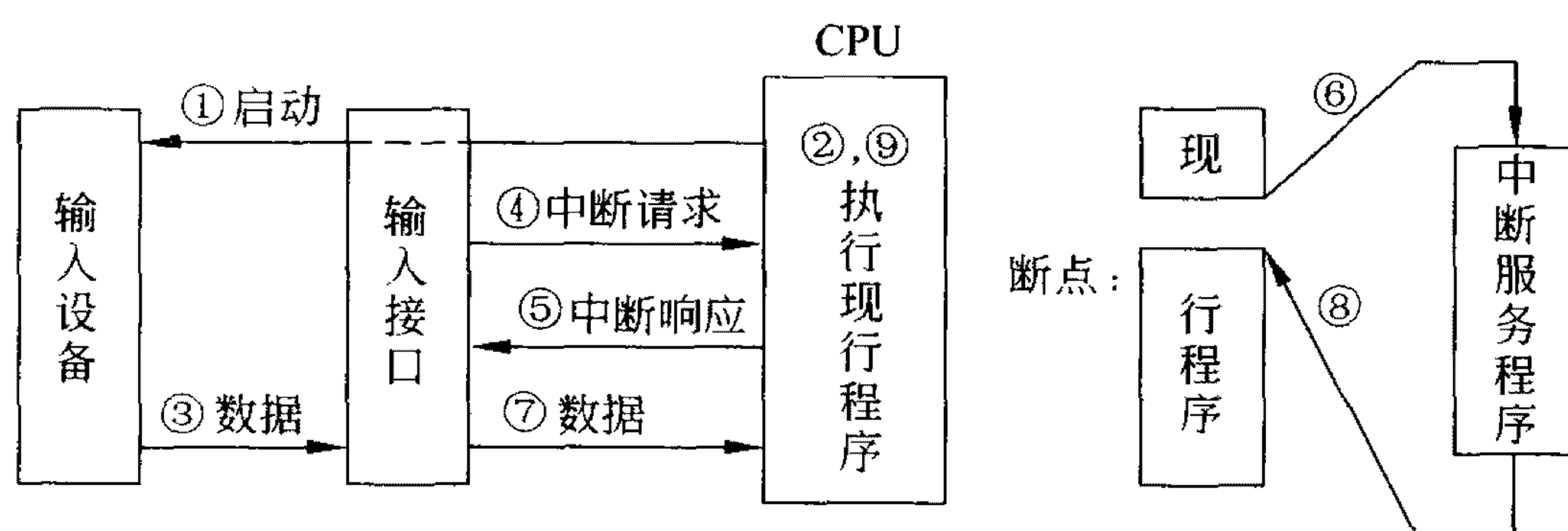


图 9-1 运用中断技术输入数据的示意图

- ① 现行程序执行时,首先通过输入接口启动输入设备工作。
- ② CPU 执行现行程序并且等待中断。
- ③ 输入设备组织了一个数据存入输入接口电路。
- ④ 接口电路向 CPU 发出“中断请求”,请求 CPU 中断现行程序。
- ⑤ CPU 满足一定条件后,向接口发出中断响应信号。
- ⑥ CPU 中断现行程序,转向预先设计好的“中断服务程序”。
- ⑦ CPU 执行服务程序,从接口电路中取走输入数据。
- ⑧ 中断服务程序执行完了,返回现行程序的断点,继续执行被中断的程序,等待第二次中断。



从以上描述的过程看出,在输入设备输入数据期间(这一过程,相对于 CPU 的速度而言是很慢的),CPU 并没有“等待”,而是执行现行程序。只有在 CPU 响应中断后,现行程序才被临时中断,CPU 抽出很短的时间去进行中断处理,随即又执行被中断的程序。从宏观上看,CPU 和外部设备是“同时”工作的。

2. 什么是中断

综上所述,CPU 暂停执行现行程序,转而处理随机事件,处理完毕后再返回被中断的程序,这一全过程称为中断。

3. 中断源

能够引发 CPU 中断的信息源,称为中断源。

由时钟系统引发的中断称为时钟中断;由外设接口引发的中断,称为外设中断;由硬件故障引发的中断,称为故障中断……

系统为了实现中断而采取的硬件和软件措施,称为中断系统。

4. 中断系统应具备的基本功能

① 为了加强中断系统的灵活性,对于硬件设备的中断请求,应当具有屏蔽和开放的性能,使得程序员能够灵活控制。

② 能实现“中断判优”即中断源排队,当有多个中断源提出请求时,能够优先响应高级别的中断。

③ 能够实现中断嵌套,即较高级别的中断源能够中断较低级别的中断服务程序。

④ 系统响应中断后,能够自动转入中断处理,中断处理结束,能自动返回。

9.2 80x86 中断指令

1. 开中断指令 STI

该指令使标志寄存器中断允许标志(I 标志)置 1,从而使 CPU 处于开中断状态,允许响应来自引脚 INTR 的中断请求。

2. 关中断指令 CLI

该指令使标志寄存器中断允许标志(I 标志)置 0,使 CPU 处于关中断状态,不响应来自引脚 INTR 的中断请求。

3. 软件中断指令 INT n

其中 n 为 $0 \sim 255$ 的无符号整数, n 称为中断类型码又称中断号。该指令的功能是调用 n 型中断服务程序。在实地址模式下,CPU 执行该指令时,完成以下操作:

① 标志寄存器内容压栈保存。

② 使标志寄存器的 T 标志置 0, 从而禁止单步操作; 使 I 标志置 0, CPU 处于关中断状态。

③ 断口地址 CS、IP 内容先后压栈。

INT n 指令的后续指令地址就是断口地址。INT n 指令取出后(还未执行时)CS: IP 的内容就是后续指令地址, 后续指令的地址称为断口地址。

④ CPU 从系统 RAM $4 \times n \sim 4 \times n + 3$ 单元取出 n 型中断向量 \rightarrow IP、CS。

⑤ CPU 根据 CS: IP 的内容转向 n 型中断服务程序。

4. 中断返回指令 IRET

该指令从栈顶弹出 6 个元素依次写入 IP、CS 和标志寄存器。

该指令是中断服务程序的出口指令, 如果在执行 IRET 之前, 栈顶依然是程序的断口地址, 那么 IRET 执行后, 就能够达到中断返回的目的, 否则不能。这就是说, 在进入中断服务程序以后, 程序员可以进行堆栈操作, 但是在执行 IRET 指令之前必须把进栈的数据全部弹出, 保证栈顶是程序的断口地址, 否则执行 IRET 将造成灾难性的后果。

9.3 中断向量

在实地址模式下, 中断向量表是使 CPU 转向中断服务程序的重要措施。

1. 中断向量

什么是中断向量呢? 实地址模式下, 中断服务程序的入口地址就是中断向量。

中断向量由两部分组成:

① 服务程序所在代码段的段基址(2 个字节)。

② 服务程序入口的偏移地址(2 字节)。

CPU 把其中的段基址乘以 16, 加上入口的偏移地址就可以最终算出服务程序入口的物理地址。

2. 中断向量表

微型计算机系统中, 最多允许有 256 种中断(实际使用中没有这么多), 为了区别它们, 给每一种中断分配一个中断号, 中断号又称中断类型码(n)。对应于每一种中断应当有一个中断服务程序。同理对应于每一种中断都有一个中断向量。

一种类型的中断, 其中断向量为 4 个字节, 256 种中断, 其中断向量总共占用 1024 个字节。在实地址模式环境下, CPU 规定中断向量集中存放在系统 RAM 最低端的 1024 个单元之中(物理地址为 00000H~003FFH), 存放中断向量的这 1024 个单元就构成了一张中断向量表。图 9-2(a)表示 n 型中断向量 4 个字节的存放规律, 以及 n 型中断向量和存放该向量的单元地址之间的关系。例如: 5 型中断向量存放在 $4 \times 5 \sim 4 \times 5 + 3$ 的 4 个单元之中。其中 14H 和 15H 单元存放 5 型服务程序入口的偏移地址, 16H 和 17H 单元存放 5 型服务程序入口的段基址, 以此类推。图 9-2(b)表明 0~255 型中断向量的排

列规律。

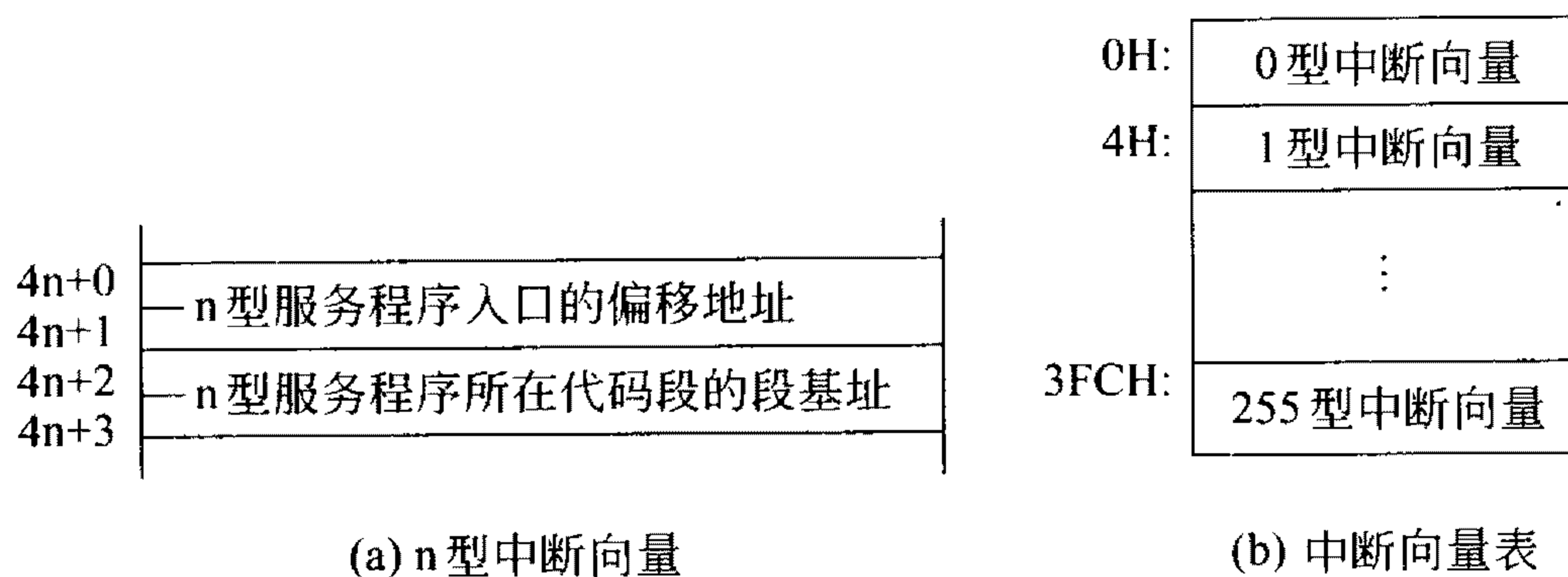


图 9-2 中断向量和中断向量表

3. 中断向量的引导作用

CPU 响应中断后，中断向量将引导 CPU 去执行相应的中断服务程序。图 9-3 形象地说明了 CPU 执行 INT 21H 指令时，中断向量的“引导”作用。

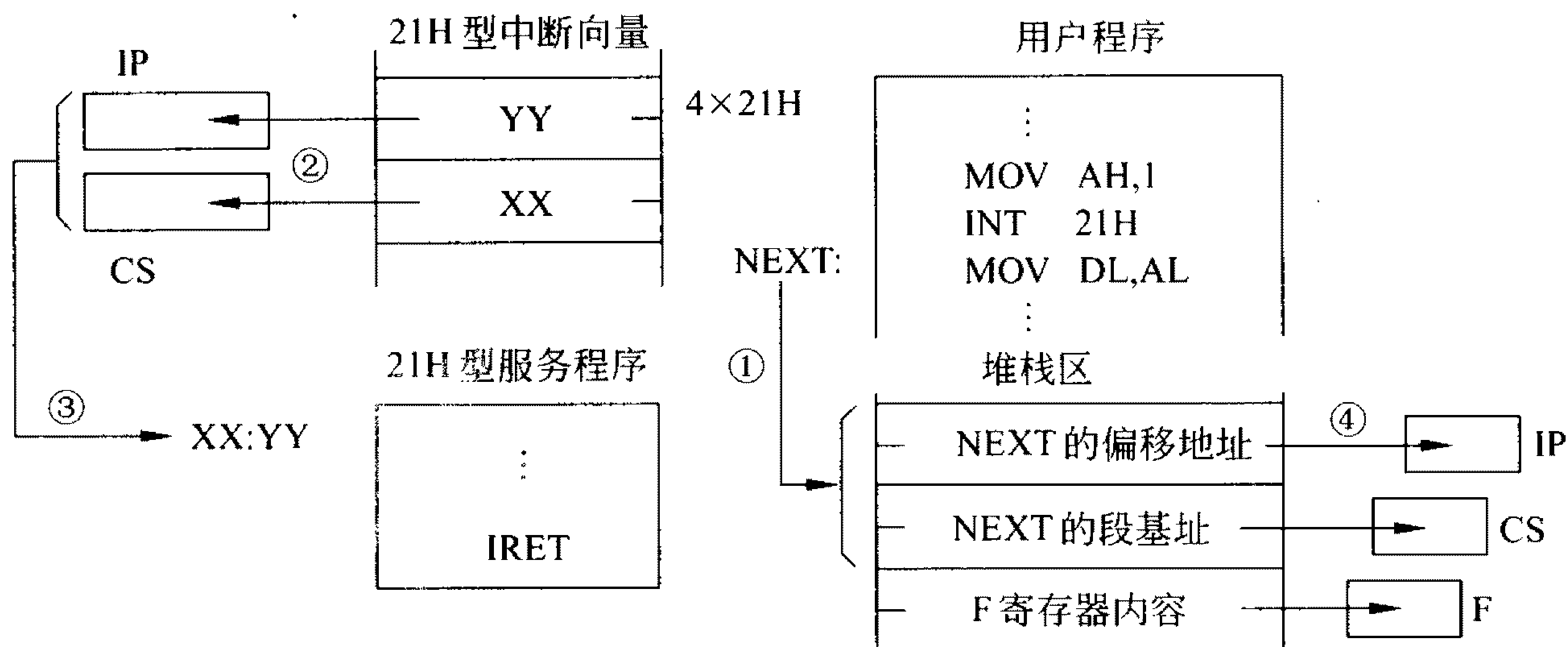


图 9-3 中断向量的引导作用

启动 DOS 之后，假设 21H 型服务程序被放在地址等于 XX:YY 开始的内存区，其中断向量 XX:YY 存放在 $4 \times 21H \sim 4 \times 21H + 3$ 的单元中。

CPU 执行用户程序，当取出 INT 21H 指令之后，CS:IP 必定等于标号 NEXT 所在单元的逻辑地址，执行 INT 21H：

- ① F 寄存器的内容、CS、IP 的当前值被压入堆栈。
- ② CPU 从 $4 \times 21H \sim 4 \times 21H + 3$ 单元中取出 21H 型中断向量写入 IP、CS 之中。
- ③ CPU 根据 CS:IP 的值转向 21H 型服务程序。
- ④ 21H 型服务程序执行完毕，执行 IRET 指令时，CPU 从栈顶弹出 NEXT 的两个分量→IP、CS，接着弹出响应中断前的标志寄存器内容→标志寄存器。
- ⑤ CPU 根据 CS:IP 返回断点 NEXT，执行“MOV DL,AL”完成中断全过程。

从上述过程可以清楚地看到：中断向量的作用就是引导 CPU 执行相应的中断服务程序。

中断向量的引导作用,启发了我们设计一条软中断指令的思路:如果用户设计了一个 60H 型中断服务程序,只要预先把这个服务程序的入口地址写入到 $4 \times 60H \sim 4 \times 60H + 3$ 单元,那么在用户程序中执行“INT 60H”,CPU 就转而执行新设计的 60H 型中断服务程序了。

4. 中断向量表的初始化

中断向量表的初始化有复杂的过程,归纳起来有以下几点:

- ① 由 BIOS 提供的服务程序,其中断向量是在系统加电后,由 BIOS 负责设置的。
- ② 由 DOS 提供的服务程序,其中断向量,在启动 DOS 时由 DOS 负责设置。
- ③ 用户程序可以开发自己的中断服务程序,用它取代系统原有的服务程序。用户开发的服务程序,其中断向量应当由用户程序本身负责设置。

DOS 系统为用户程序读取和写入中断向量,设计了 2 个子程序。

【INT 21H 的 35H 号子功能】

功能:读取中断向量。

入口参数:AH=35H,AL=中断类型码(即中断号)。

出口参数:ES:BX=中断向量的逻辑地址。

【INT 21H 的 25H 号子功能】

功能:写入中断向量。

入口参数:AH=25H,AL=中断类型码(即中断号)。

DS:DX=要写入的中断向量,即:

DS=中断服务程序所在代码段的段基址。

DX=中断服务程序入口的偏移地址。

出口参数:无。

这 2 个子程序对于设计中断程序是十分有用的,怎样调用它们呢?我们将在以后的中断程序设计中详细介绍。

5. 关于中断向量表的说明

在实地址模式下,系统 RAM 最低端的 1024 个单元,为中断向量表。但是并非每一个表项都是中断向量。BIOS 利用某些表项做为“参数指针”,参数指针指向的是一群参数,而不是中断服务程序,因此称它们为“向量”是比较合适的。

例如:

1DH 型向量,指向屏幕参数表。

1EH 型向量,指向软盘参数表。

1FH 型向量,指向图型字符表。

41H 型向量,指向第一台硬盘参数表。

46H 型向量,指向第二台硬盘参数表。

用户程序不能改动这些向量,当然也不能执行以 1DH,1EH,1FH,41H,46H 为中断类型码的软中断指令,否则系统将会瘫痪。

9.4 微型计算机系统的中断分类

在实地址模式下,按中断源划分,系统中断可以分为:CPU 中断,硬件中断和软件中断 3 类,本节先介绍 CPU 中断和软件中断。

9.4.1 CPU 中断

CPU 中断是指:CPU 执行某些操作而引发的中断,这类中断使用了 0、1、3、4、6、7 中断号。

1. 除法错中断——0 型中断

CPU 执行 DIV 或 IDIV 指令,如果除数为 0,或者商数超出寄存器的表示范围,CPU 自动调用 0 型中断服务程序。

DOS 为 0 型中断设计的服务程序并没有为“除法错”采取什么补救措施,仅仅是显示一行错误信息“Divide overflow”,然后返回 DOS。因此,用户程序执行“INT 0”指令是毫无意义的。

2. 单步中断——1 型中断

当标志寄存器的 T 标志为 1 时,CPU 一条指令执行完毕,自动调用 1 型中断服务程序。但是,DOS 为 1 型中断设计的服务程序只有一条 IRET 指令。实际上,单步中断是为 DEBUG 程序的需要而设计的。在用户程序中执行“INT 1”指令是毫无意义的。

3. 断点中断——3 型中断

CPU 执行“INT 3”指令后,调用 3 型中断服务程序,而 DOS 系统的 3 型中断服务程序也只有一条 IRET 指令。断点中断也是为 DEBUG 程序的需要而设计的。而在 DEBUG 程序当中,利用“INT 3”指令设置程序断点,当执行到“INT 3”指令时,转入 3 型中断服务程序(该程序应由 DEBUG 设计),显示断点前程序的执行结果。在用户程序中执行“INT 3”指令是毫无意义的。

4. 溢出中断——4 型中断

对应的软件中断指令有两种汇编格式,功能稍有不同。

① INTO

② INT 4

当 FLAG 寄存器的溢出标志为 1,在这种条件下,执行“INTO”指令,将会调用 4 型服务程序。否则,如果溢出标志为 0,执行“INTO”指令是无效的。

无论溢出标志为 0,还是为 1,执行“INT 4”指令都将调用 4 型服务程序。这是两种指令格式在功能上的差别。

但是和 1 型、3 型中断一样,DOS 为 4 型中断设计的服务程序也只有一条 IRET 指令。如果用户程序需要对运算过程进行监控,应当在有可能产生溢出的运算操作之后安排一条 INTO 指令监测溢出标志,同时,还要自行设计溢出处理程序取代原先的 4 型中断服务程序。

9.4.2 软件中断

执行有定义的 INT n 指令而引发的中断,称为软件中断。在这里,之所以加了“有定义的”这一限制词,是因为并非所有的中断号都有与之配套的中断服务程序。

软件中断使用 05H,10H~FFH 中的若干个中断号。软件中断又可分为 BIOS 中断、DOS 中断。

1. BIOS 中断

BIOS 中断,占用了 05H、10H~1FH 中断号,用户程序执行相关的软中断指令可以调用相应的中断服务程序。

INT 05H 屏幕打印;
INT 10H 屏幕显示 I/O;
INT 11H 设备配置检测;
INT 12H 测试内存容量;
INT 13H 磁盘 I/O;
INT 14H 串行通信 I/O;
INT 15H BIOS 扩展功能;
INT 16H 键盘 I/O;
INT 17H 打印机 I/O;
INT 18H 启动 PC 机 ROM BASIC(AT 机)无;
INT 19H 重新装入引导程序;
INT 1AH 实时时钟管理。

有 3 个问题需要说明:

① INT 1BH 调用 Ctrl-Break 处理程序。

BIOS 为 1BH 中断设计的服务程序只有一条 IRET 指令。启动 DOS 之后,DOS 用自己的 Ctrl-Break 处理程序取代了它。用户程序不能执行 INT 1BH。

② INT 1CH

1CH 中断服务程序只有一条 IRET 指令,该服务程序由系统的 08 型中断每隔 55ms 调用一次。从这个意义上讲,1CH 中断是系统时钟中断的“外扩”,用户可以设计自己的定时操作程序,取代原先的 1CH 中断服务程序。

③ 中断号 1DH,1EH,1FH,41H,46H 也被 BIOS 占用了。但是,与这些中断号对应的并不是中断服务程序。也就是说,不存在与这些中断号对应的软中断指令,用户程序如果执行 INT 1DH~INT 1FH,INT 41H,INT 46H,必将引起系统瘫痪(参见中断向量表的说明)。

2. DOS 中断

DOS 中断,又分为 DOS 专用中断,DOS 保留中断,用户可调用的 DOS 中断以及保留给用户开发的中断。

(1) DOS 专用中断

① 22H 型中断 程序正常结束时,DOS 将自动调用该中断返回父进程。

② 23H 型中断 程序非正常结束时(如:用户按下 Ctrl+C,或者 Ctrl+Break 中途停止程序的运行),DOS 调用该中断。

③ 24H 型中断 程序运行发生严重错误时(例如:对软磁盘文件进行操作的时候,驱动器小门没有关闭,或进行打印操作而打印机没有连通),DOS 自动调用此类中断,发出错误信息“Not ready...Abort,Retry,Ignore?”,这些中断是 DOS 专用的,DOS 在调用此类中断之前,还要做些准备工作。因此,用户程序不能直接调用这些中断。

④ 28H~3FH 型中断 也为 DOS 专用。

(2) 用户可调用的 DOS 中断

① 20H 型中断 用户程序执行 INT 20H 指令可结束程序的运行,返回 DOS。但必须注意:在执行 INT 20H 之前,必须保证用户程序 CS 寄存器的内容等于 PSP 段基址。因此在用户的 COM 文件中,可以直接使用 INT 20H 返回 DOS。

INT 20H 与 INT 21H 的 0 号功能调用,完成相同的操作。

② 21H 型中断 DOS 系统的许多功能都集中在 21H 型中断服务程序中。用户程序把功能号写入 AH 寄存器,设置相应的入口参数,然后执行 INT 21H 即可调用不同的功能。我们把执行 INT 21H 指令所完成的功能,称为“DOS 系统功能调用”。

③ 25H 型中断 此类中断在指定的驱动器上,按照扇区号读取信息。

④ 26H 型中断 在指定的驱动器上,按照扇区号写入信息。INT 25H,INT 26H 称为“绝对磁盘读写调用”。在此类调用中,DOS 系统不使用文件控制块,也不使用文件号去管理磁盘文件,而是按照扇区号直接进行磁盘信息的读/写。

⑤ 27H 型中断 中止并驻留程序于内存之中。用户程序执行 INT 27H 可以中止程序的运行,并且把欲驻留的程序段驻留在内存之中。

(3) 用户可开发的 DOS 中断

60H~66H 是保留给用户使用的中断号。DOS 没有为它们设计服务程序。启动 DOS 后,60H~66H 型“中断向量”均为 0 值。正因为如此,如果没有开发出 60H~66H 型中断服务程序,没有改写 60H~66H 型中断向量,千万不可调用此类中断,否则系统瘫痪。

(4) DOS 保留的中断

DOS 为了自身版本的升级和功能的扩充,保留了若干个中断号,它们是:42H~45H,4BH~5FH,68H~6FH,72H~74H,77H~7FH。

以上关于软件中断的概念,在设计应用程序时是十分有用的,实际上,如果不涉及 BIOS 中断,不涉及 DOS 中断,用户将无法设计应用程序。

9.5 8259A 中断控制器

硬件中断是由 CPU 以外的器件发出的中断请求信号而引发的中断。80x86CPU 只有 2 个引脚(INTR 和 NMI)可以接收外部的中断请求脉冲,为了管理众多的外部中断源,Intel 公司设计了专用的配套芯片——8259A 中断控制器。

8259A 是可编程芯片,一片 8259A 管理 8 级中断源,两片 8259A 级连管理 15 级中断。

本节首先介绍 8259A 的内部结构、工作原理和中断管理方式,为后续章节介绍系统硬件中断打好基础。

9.5.1 8259A 内部结构

8259A 内部结构如图 9-4 所示。

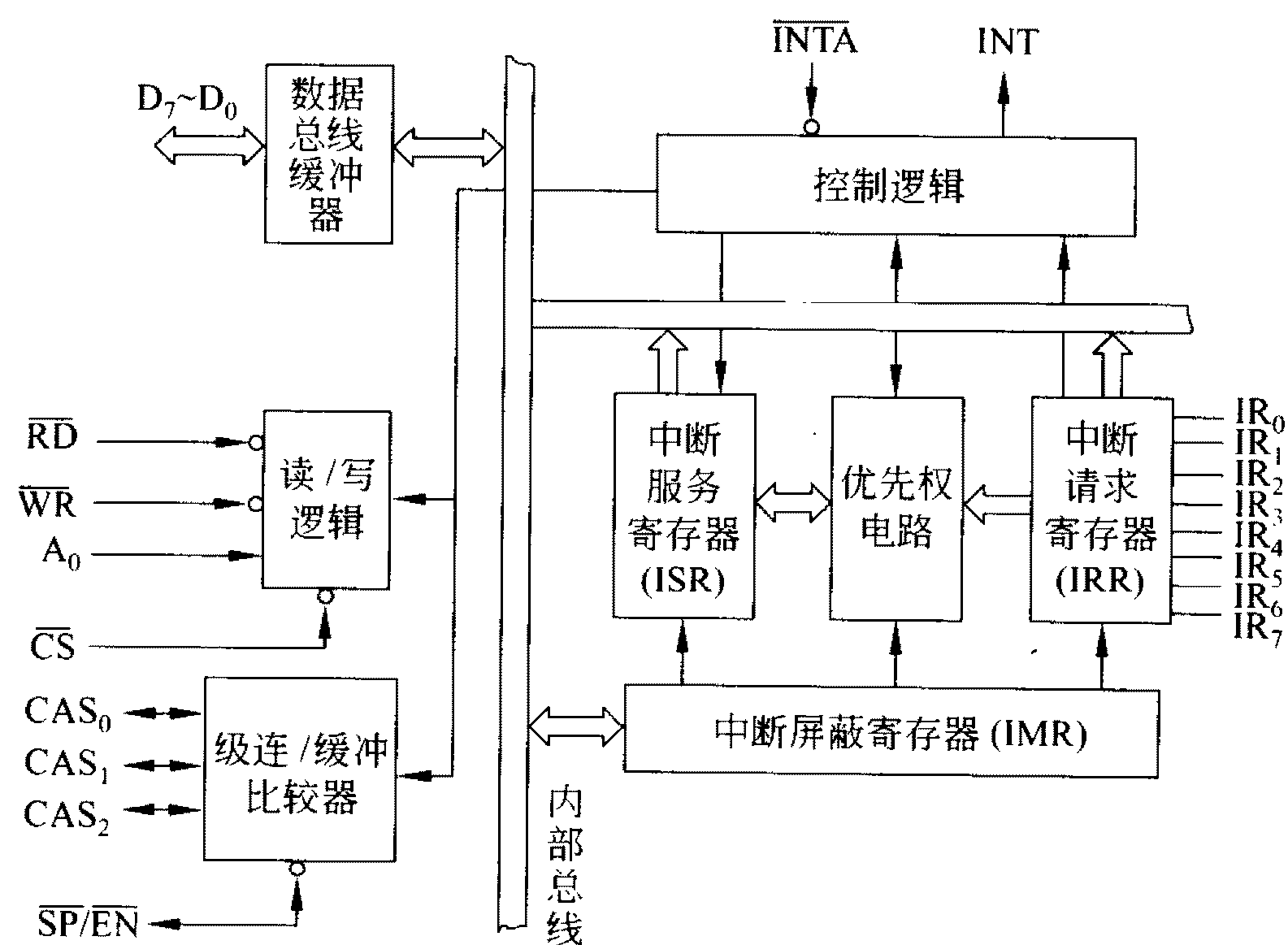


图 9-4 8259A 内部结构

1. 8259A 的内部结构

(1) 中断请求寄存器(8 位) IRR

它寄存引脚 $IR_0 \sim IR_7$ 的中断请求,当 IR_i 中断请求信号有效时, IRR_i 位置 1。 IRR_7 位与引脚 IR_7 对应,其余类推。

(2) 中断屏蔽寄存器(8 位) IMR

它寄存程序员写入的中断屏蔽字(即操作命令字 OCW_1)。 IMR_i 位与 IRR_i 位一一对应,如果 IMR_i 位为 1,锁存在 IRR_i 位的中断请求就不能送达优先权电路。例如:CPU 写入的屏蔽字为 00000100,那么 IR_2 的中断请求就被屏蔽了,而其他各级中断仍被允许。

(3) 优先权电路

该电路负责判别同时送达优先权电路的中断源,哪一个级别最高,它还要判别 CPU 正为之服务的中断源和新进入优先权电路的中断源,哪一个级别更高。

(4) 中断服务寄存器(8 位) ISR

ISR_i 位和 IR_i 中断源一一对应,它记录 CPU 正为之服务的是哪一个中断源。

假如经过优先权电路的判别,IR₆ 中断被选中,那么,8259A 向 CPU 提出中断请求。8259A 在收到第一个中断响应信号时,使 ISR₆ 位置 1、IRR₆ 位清零。ISR₆ 位置 1,表明 CPU 正在处理 IR₆ 中断。

如果 IR₀ 的中断请求中断了 IR₆ 的服务程序,那么 ISR₀ 也置为 1。ISR₀ 置 1、ISR₆ 置 1,表明 IR₀、IR₆ 的服务程序都没执行完。从这个意义上讲,ISR 的每一位都是“中断服务标志位”。ISR 的置 1 位一直要保持到该中断源的中断处理结束,当收到中断结束命令后,该位置 0。反之,ISR 的置 1 位清零,表明该位所对应的中断源处理程序已经结束。

(5) 数据总线缓冲器

它是 8259A 与系统数据线的接口模块,是 8 位的双向三态缓冲器。对 8259A 初始化编程时写入的命令字,以及 8259A 的状态信息都是通过它写入和读出的。

(6) 读写控制模块

该模块接收片选信号 \overline{CS} 、端口选择信号 A₀ 和读写控制信号 \overline{RD} 、 \overline{WR} 。

一片 8259A 在系统中占用两个端口地址,用末位地址线 A₀ 选择端口,其他地址线通过译码产生 8259A 的片选信号。

PC 机使用两片 8259A 级连,主 8259A 端口地址为 20H、21H,从 8259A 端口地址为 A0H、A1H。

(7) 控制电路

控制电路是 8259A 内部的控制器。它有一组寄存初始化命令字(ICW₁~ICW₄)的寄存器和一组寄存操作命令字(OCW₁~OCW₃)的寄存器,以及相关的控制逻辑。这些命令字寄存器通过译码产生内部控制信号,并根据中断请求,中断判优的结果,通过 INT 端向 CPU 提出中断请求,通过 \overline{INTA} 端从总线控制器接收中断响应信号。在 80x86 系统中,中断响应信号为 2 个连续的负脉冲。

(8) 级连缓冲/比较器

该模块在级连方式的主-从结构中,用来存放和比较从 8259A 的设备代码。CAS₂~CAS₀ 为级连信号线,在主-从结构中,主、从 8259A 的 CAS₂~CAS₀ 对应连接。主 8259A 的 CAS₂~CAS₀ 为输出线,从 8259A 的 CAS₂~CAS₀ 为输入线。

$\overline{SP/EN}$ 为双向双功能引脚。工作在级连方式的时候,主 8259A 的 $\overline{SP/EN}$ 接高电平,从 8259A 的 $\overline{SP/EN}$ 接低电平。

2. 8259A 的中断过程

8259A 的中断过程,就是微型计算机系统响应可屏蔽中断的过程,这一过程,简单描述如下:

① 首先由中断请求寄存器寄存加到引脚 IR₀~IR₇ 上的中断请求。

- ② 在中断屏蔽寄存器的管理下,没有被屏蔽的中断请求被送到优先权电路判优。
- ③ 经过优先权电路的判别,选中当前级别最高的中断源,然后从引脚 INT 向 CPU 发出中断请求信号。
- ④ CPU 满足一定条件后,向 8259A 发出 2 个中断响应信号(负脉冲)。
- ⑤ 8259A 从引脚 $\overline{\text{INTA}}$ 收到第 1 个中断响应信号之后,立即使中断服务寄存器中与被选中的中断源对应的那一位位置 1,同时把中断请求寄存器中的相应位清零。
- ⑥ 从引脚 $\overline{\text{INTA}}$ 收到第 2 个中断响应信号后,8259A 把选中的中断源类型码 n ,通过数据线送往 CPU。
- ⑦ 在实地址模式下,CPU 从 $4 \times n \sim 4 \times n + 3$ 单元取出该中断源的中断向量 $\rightarrow \text{IP}$ 、 CS ,从而引导 CPU 执行该中断源的中断服务程序。

9.5.2 8259A 中断管理方式

8259A 的中断管理十分灵活,它涉及 5 个主要方面,即:中断触发方式、中断屏蔽方式、优先级管理方式、中断结束方式和总线连接方式。每一种方式又有几种模式可供选择。如图 9-5 所示。

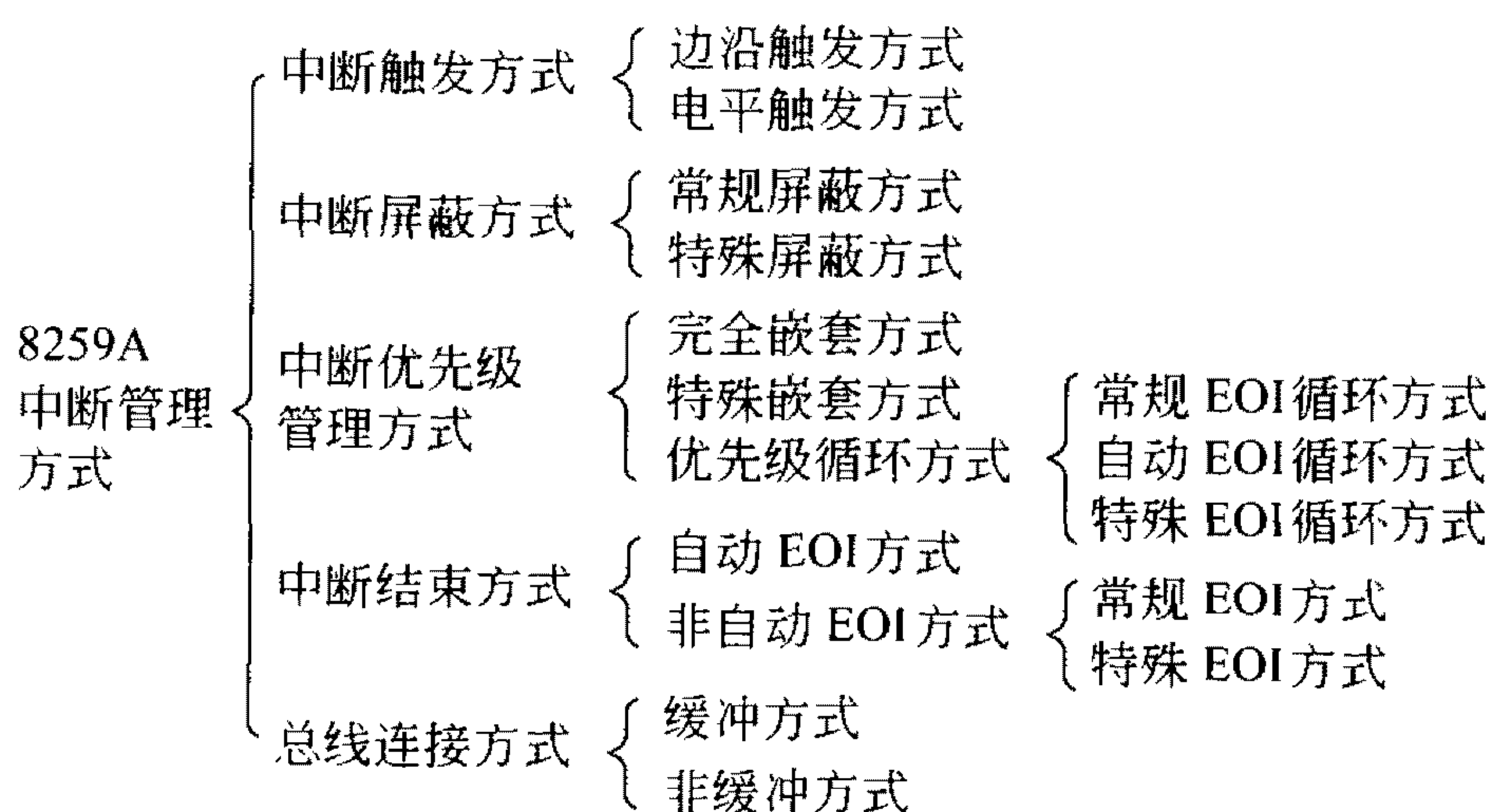


图 9-5 8259A 中断管理方式

下面将着重介绍各种方式的含义。

1. 中断触发方式

8259A 有两种中断触发方式:

(1) 电平触发方式

当 $\text{IR}_0 \sim \text{IR}_7$ 出现高电平时,表示有中断请求。

高电平要持续多长时间? 以 IR_0 为例, IR_0 的高电平必须保持到 CPU 响应 IR_0 中断,而且保持到 8259A 收到第一个中断响应脉冲之前,否则 IR_0 的本次中断可能被丢失,8259A 自动认为是 IR_7 有中断请求。当 IR_0 的中断服务结束, ISR_0 位被清零之前, IR_0 的高电平必须撤销,否则可能引起第二次中断。

(2) 边沿触发方式

$\text{IR}_0 \sim \text{IR}_7$ 出现低电平到高电平的跃变,表示有中断请求。在 CPU 响应中断,8259A

收到第一个中断响应脉冲之前,同一个输入端不应当出现第二次低电平到高电平的跃变,否则第一次跃变(即中断请求)可能被丢失。

在 80x86 微型计算机系统中采用边沿触发方式。

2. 中断屏蔽方式

8259A 有两种中断屏蔽方式。

(1) 常规屏蔽方式

将中断屏蔽寄存器的某一位置 1,即可屏蔽相应级别的中断请求。例如在初始化编程之后,在主程序或者某个服务程序中把 11110000 写入中断屏蔽寄存器,即可屏蔽 $IR_7 \sim IR_4$ 的中断请求,而开放 $IR_3 \sim IR_0$ 中断。

(2) 特殊屏蔽方式

通常情况下,当一个中断被响应时,禁止同级和较低级别的中断请求。而特殊屏蔽方式的功能是:当一个中断被响应时,仅仅屏蔽同级别的再次中断,而允许较低或者较高级别的中断源中断正在执行的服务程序。

在 80x86 微型计算机系统中采用常规屏蔽方式。

3. 中断优先级管理方式

8259A 对中断源的优先级管理有两种主要方式。

(1) 完全嵌套方式

完全嵌套方式,又称固定优先级方式。完全嵌套方式的特点是:

- ① $IR_0 \sim IR_7$ 的优先级是固定的, IR_0 为最高级, IR_1 次之,……, IR_7 为最低级。
- ② CPU 响应某一级中断时,8259A 将 ISR 中与该中断源对应的那一位位置 1,并自动禁止同级和较低级的中断请求。

(2) 优先级循环方式

$IR_0 \sim IR_7$ 的中断级别不是固定的。当任一中断服务程序结束后,该中断源就自动降为最低级,而原先比它低一级的中断源自动升为最高级。

中断处理结束时,向 8259A 发出一个兼有优先级循环功能的中断结束命令就可实现优先级自动循环。在 80x86 微型计算机系统中采用完全嵌套方式。

4. 中断结束方式

把 ISR 寄存器中断服务标志 ISR_i 位清零,这就意味着结束 i 中断源的中断服务。怎样才能使 ISR_i 位清零呢? 8259A 提供了 6 种中断结束方式,其中有 3 种方式兼有优先级循环功能,另外 3 种不兼有优先级循环功能。

(1) 自动 EOI 方式

自动中断结束方式的功能是:8259A 在收到第二个中断响应信号之后,自动把 ISR 中置 1 的最高优先级服务标志 ISR_i 位清零。采用这种方式,不需要在 IR_i 服务程序中安排常规中断结束命令。

由于自动 EOI 方式是在响应中断后,执行 IR_i 服务之前,“提前”使 ISR_i 位清零。因

此,必然带来一个新的问题,即在 IR_i 服务程序执行过程中,如果有新的中断请求,不论它的级别高低,只要 CPU 是处于开中断状态,都将中断 IR_i 服务程序。因而可能出现低级或同级中断源中断高级服务程序的不合理现象,这种现象称为“重复嵌套”。为了防止“重复嵌套”,服务程序必须在关中断的前提下执行。

(2) 常规 EOI 方式

初始化编程时,选择“非自动中断”方式。然后在服务程序结束,执行 IRET 指令之前,向 8259A 送一个“常规中断结束”命令字。8259A 收到常规中断结束命令字之后,把 ISR 中优先级最高的置 1 位清零。

在全嵌套方式下,中断级别是固定的,8259A 总是响应优先级最高的中断。保存在 ISR 中的,最高优先级置 1 位,肯定对应正在执行的服务程序。反过来说,正在执行的服务程序发出的常规 EOI 命令,它所清零的 ISR_i 位,一定是与该中断源对应的中断服务标志,所以在全嵌套方式下,就应当使用常规 EOI 命令,结束中断。

但是,在采用特殊屏蔽方式的时候,就不能使用常规 EOI 命令。原因很明显,因为特殊屏蔽方式下,正在执行的高级中断源服务程序可能被“挂起”,而响应新的低级中断。显然,如果在低级服务程序中发出常规 EOI 命令,它所清零的不是本身的 ISR_i 位,而是级别比它高的 ISR_i 置 1 位。

工作在特殊屏蔽方式时,应使用特殊中断结束命令。

(3) 特殊中断结束方式

初始化编程的时候,选择“非自动中断”方式,然后在服务程序结束,执行 IRET 指令之前,向 8259A 送一个“特殊中断结束”命令字。8259A 根据命令字 $L_2 \sim L_0$ 位的编码,把 ISR 中的指定位清零。特殊中断结束命令,可以在任何情况下使用。

(4) 自动 EOI, 优先级循环方式

初始化编程,选择自动中断结束方式,并且向 8259A 写入“设置自动 EOI, 优先级循环”的操作命令 OCW_2 , 即可选择这种方式。在 CPU 响应中断,8259A 得到第二个中断响应信号时,自动把 ISR 中置 1 的最高优先级 ISR_i 位清零,并且完成优先级循环。

(5) 常规 EOI, 优先级循环方式

初始化编程选择“非自动中断”方式,然后在 IR_i 服务程序结束,执行 IRET 之前,安排一条“常规 EOI, 优先级循环”的命令字,送往 8259A。8259A 收到此命令后,把 ISR 中最高优先级置 1 位清零,同时完成优先级循环。

(6) 特殊 EOI, 优先级循环方式

初始化编程选择“非自动中断方式”,中断服务结束,执行 IRET 之前,向 8259A 写入“特殊 EOI, 优先级循环”的操作命令 OCW_2 , 即可将指定的 ISR_i 位清零,并完成优先级循环。

80x86 微型计算机系统采用常规中断结束方式。

5. 总线连接方式

8259A 数据线与系统数据总线的连接有两种方式。

(1) 缓冲方式

如果 8259A 通过总线驱动器和系统数据线相连,此时 8259A 应选择为缓冲方式。定义为缓冲方式之后, $\overline{SP}/\overline{EN}$ 即为输出端。在 8259A 输出中断类型码的时候, $\overline{SP}/\overline{EN}$ 输出一个低电平,用此信号可做为总线驱动器的启动信号。

(2) 非缓冲方式

如果 8259A 数据线与系统数据线直接相连,那么 8259A 工作在非缓冲方式。

9.5.3 8259A 初始化

8259A 是可编程中断控制器,它的工作方式、操作模式是由写入的命令字确定的。命令字分两类:一类是初始化命令字 $ICW_1 \sim ICW_4$,另一类是操作命令字 $OCW_1 \sim OCW_3$ 。一片 8259A 有两个端口地址,由片选信号和端口选择线 A_0 (它通常接至 CPU 地址线 A_0)共同确定, $A_0=0$ 为偶地址端口, $A_0=1$ 为奇地址端口,各命令字写入的端口也有规定。

1. 8259A 初始化命令字

(1) ICW_1

ICW_1 格式如图 9-6 所示。

- ① ICW_1 写入 8259A 的偶地址端口。
- ② $D_7 \sim D_5$ 位:当 8259A 应用于 8088/8086 系统时,该 3 位无效,通常以 0 填充。
- ③ D_4 位:是 ICW_1 的标志位。 D_4 位=1 表明是 ICW_1 命令字。
- ④ D_3 位:选择中断请求信号的触发方式。 D_3 位=1 为电平触发, D_3 位=0 为边沿触发。
- ⑤ D_2 位:对于 8088/8086 系统无效,以 0 填充。
- ⑥ D_1 位:表明 8259A 的应用情况, D_1 位=1 表示系统中只有一片 8259A, D_1 位=0 表明使用多片 8259A 级连。
- ⑦ D_0 位:表示初始化编程时,是否需要写入 ICW_4 ,对于 8088/8086 系统, D_0 必须置 1,表明需要写入 ICW_4 。

(2) ICW_2

ICW_2 格式如图 9-7 所示。



图 9-6 ICW_1 格式

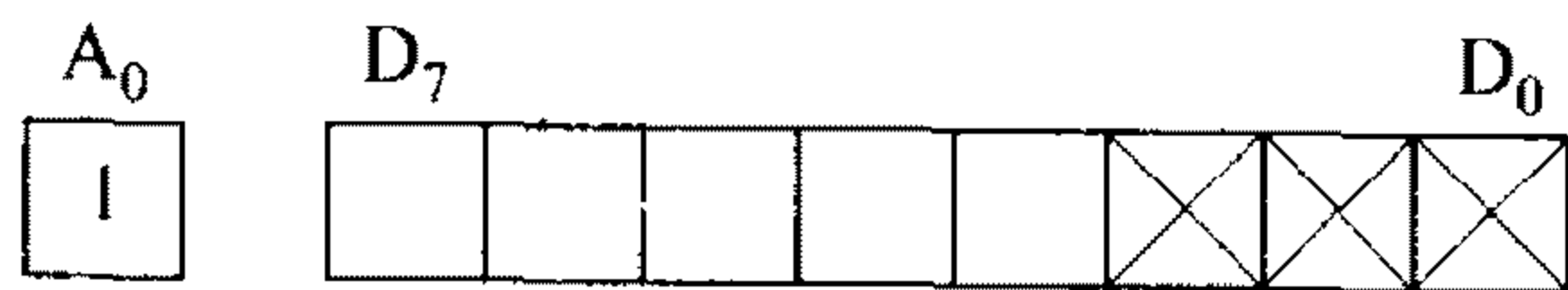


图 9-7 ICW_2 格式

- ① ICW_2 紧接着 ICW_1 写入 8259A 的奇地址端口。
- ② 对于 8088/8086 系统, ICW_2 是预置的中断类型码,初始化编程时 8259A 只接收高 5 位,形成 ICW_2 的 $D_7 \sim D_3$ 。 $D_2 \sim D_0$ 取决于当前响应的是 $IR_0 \sim IR_7$ 中的哪一个中断,再由 8259A 内部自动产生。8259A 在收到第二个中断响应信号之后,把组合之后的 ICW_2 送给 CPU。例如:初始化编程时预置的 $ICW_2=08H$,当前响应的中断是 IR_4 ,那

么,响应中断后 8259A 送出的中断类型码就是 0CH。

(3) ICW₃

当系统使用多片 8259A 级连时,才需要写入 ICW₃。写入主 8259A 和从 8259A 的 ICW₃ 的格式不同。

主 8259A ICW₃ 格式如图 9-8 所示。

从 8259A ICW₃ 格式如图 9-9 所示。

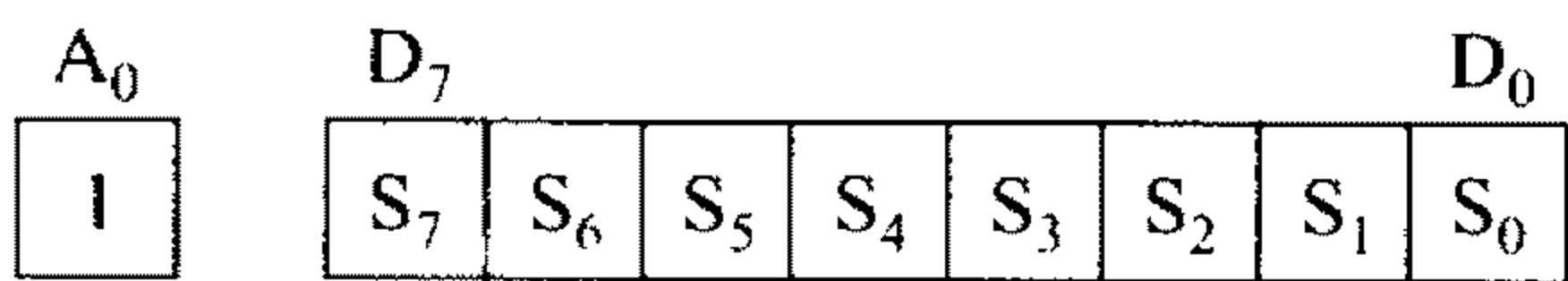


图 9-8 主 8259A ICW₃ 格式

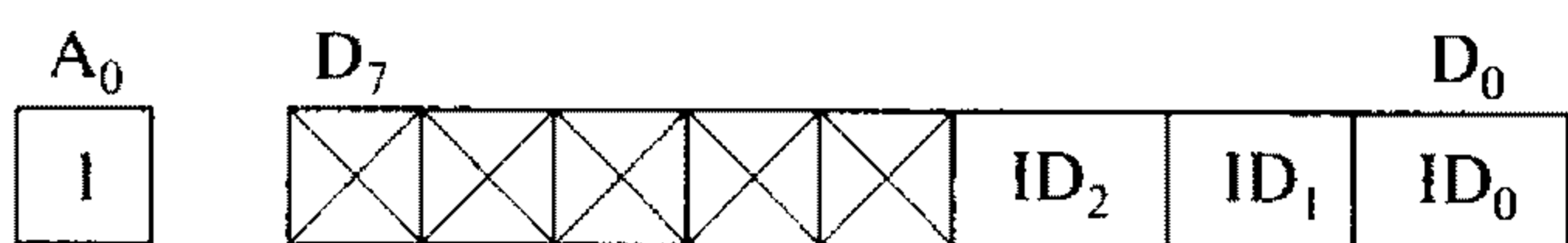


图 9-9 从 8259A ICW₃ 格式

- ① 需要时,ICW₃ 紧接着 ICW₂ 写入 8259A 的奇地址端口。
- ② 在级连方式下,从 8259A 的 INT 端要接主 8259A 的 IR₀~IR₇ 中断输入端,具体接到哪一个中断输入端? 怎样反映连接关系? 应当在主、从 8259A 初始化时,分别用 ICW₃ 进行说明。
- ③ 主 8259A ICW₃ 的 S_i 位=0,表示主 8259A 的 IR_i 端没有接从 8259A,反之 S_i 位=1,表示主 8259A 的 IR_i 端接有从 8259A。
- ④ 从 8259A 用 ICW₃ 的 ID₂~ID₀ 编码,表明该从 8259A 接到主 8259A 的哪一个中断输入端。举例来说,如果仅有一片从 8259A 的 INT 接到主 8259A 的 IR₂ 端,那么,主 8259A ICW₃ 应等于 00000100,从 8259A 的 ICW₃ 应等于 00000010。
- ⑤ 从 8259A ICW₃ 的 D₇~D₃ 不用,D₂~D₀ 位即 ID₂~ID₀ 编码称为从片的“设备代码”。

(4) ICW₄

ICW₄ 格式如图 9-10 所示。

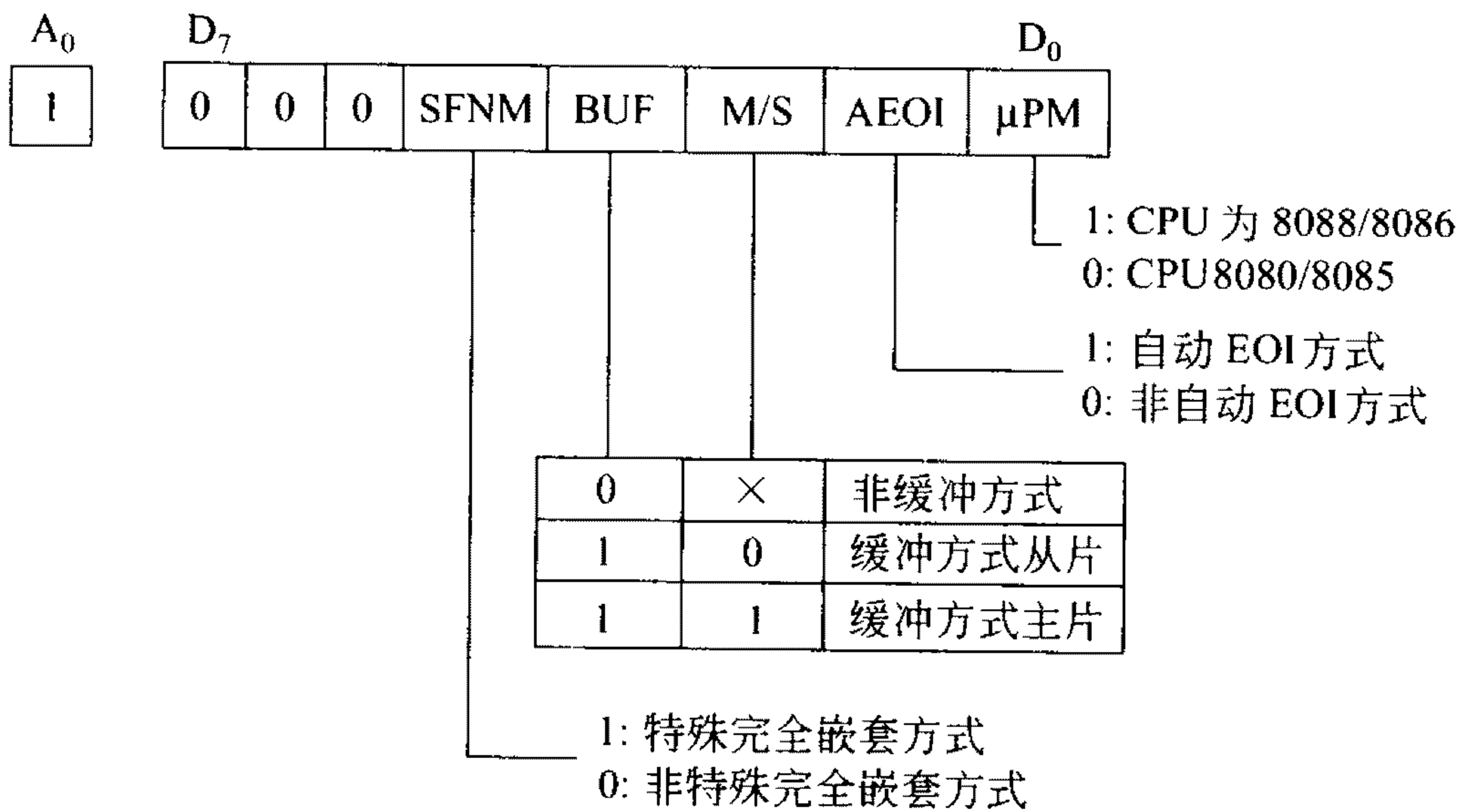


图 9-10 ICW₄ 格式

- ① ICW₄ 写入 8259A 奇地址端口。
- ② D₇~D₅ 位: 固定为 0,是 ICW₄ 的标志码。
- ③ D₄ 位: 为 1 定义为特殊完全嵌套方式。D₄ 位为 0 即选择非特殊完全嵌套方式。

④ D_3 位：用来表明数据线的连接方式。如果 8259A 通过总线驱动器连接到系统数据总线，则 BUF 位应置 1，选择缓冲方式。如果 8259A 直接和数据总线相连，则 BUF 位应置 0，选择非缓冲方式。

⑤ D_2 位：在缓冲方式下，M/S 位确定该片是主片还是从片。在非缓冲方式下，M/S 位无效。

⑥ D_1 位：选择中断结束方式。AEOI 位=1，选择自动中断结束方式；AEOI=0，选择常规 EOI 方式或特殊 EOI 方式。

⑦ D_0 位：表示 8259A 的应用环境。应用于 8088/8086 系统，该位必须置 1。

2. 8259A 操作命令

8259A 经过初始化编程之后，进入设定的工作状态，准备好接收 $IR_0 \sim IR_7$ 的中断请求。在运行过程中，用户根据需要可以写入操作命令，进一步对其进行控制。操作命令有 3 个，即 $OCW_1 \sim OCW_3$ 。

(1) OCW_1

OCW_1 称为中断屏蔽字，它写入奇地址端口，进入中断屏蔽寄存器 IMR。屏蔽字每一位 M_i 与中断请求寄存器 IRR 的每一位一一对应， M_i 位=1，将屏蔽 IRR_i 的中断请求进入优先权电路， M_i 位=0 意味着开放 IR_i 的中断， OCW_1 格式如图 9-11 所示。

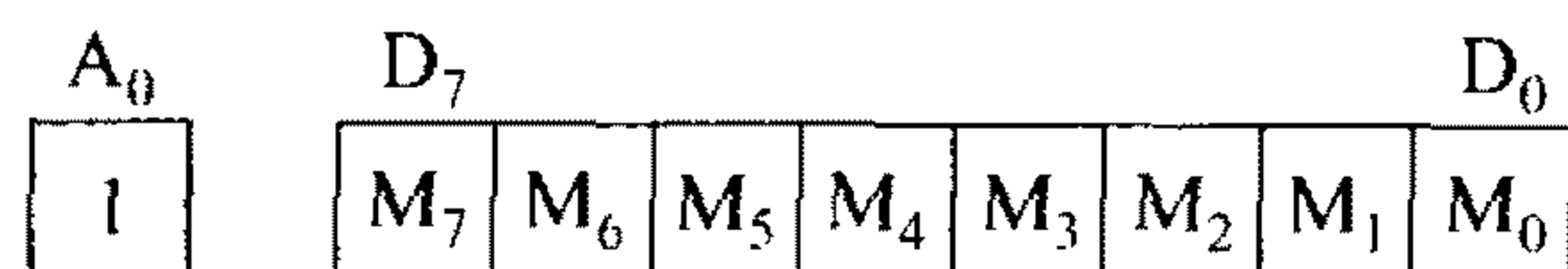
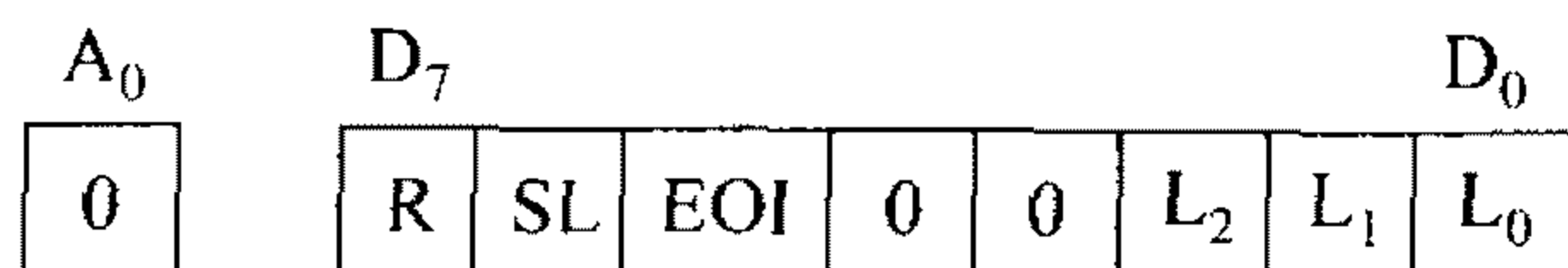


图 9-11 OCW_1 格式

(2) OCW_2

OCW_2 称为中断结束命令字，用来控制中断结束，改变优先权排序结构。该命令字写入偶地址端口， OCW_2 格式如图 9-12 所示。



R	SL	EOI	功能	L_2	L_1	L_0	对应
0	0	1	常规 EOI	0	0	0	IR_0
0	1	1	特殊 EOI	0	0	1	IR_1
1	0	1	常规 EOI，优先级循环	0	1	0	IR_2
1	1	1	特殊 EOI，优先级循环	0	1	1	IR_3
0	0	0	取消自动 EOI 时的优先级循环	1	0	0	IR_4
0	1	0	无操作	1	0	1	IR_5
1	0	0	设置自动 EOI 时的优先级循环	1	1	0	IR_6
1	1	0	指定 $IR_{L_2 \sim L_0}$ 为最低优先级	1	1	1	IR_7

图 9-12 OCW_2 格式

① OCW_2 是组合命令字， $D_4 D_3$ 位=00 是该命令字的标志。 OCW_2 应写入偶地址端口。

② R 位为优先级循环控制位，R 位为 1，具有优先级循环功能，R 位为 0 是固定优先级。

- ③ EOI 位为中断结束控制位,EOI 位为 1 具有中断结束功能。
- ④ $L_2 \sim L_0$ 编码对应于 $IR_0 \sim IR_7$ 中断源,仅当 SL 位为 1 时有效(无操作功能项除外)。
- 例如:当 $R、SL、EOI=111, L_2 \sim L_0=101$ 时,该命令字将中断服务寄存器 ISR_5 位清零,从而结束 IR_5 中断处理,并且指定 IR_5 为最低优先级。当 $R、SL、EOI=001$ 时,为常规中断结束命令,它使得 ISR 寄存器中优先级最高的置 1 位清零,从而结束当前的中断处理。

最常用的常规 EOI 命令字为 20H。

(3) OCW₃

OCW₃ 格式如图 9-13 所示。

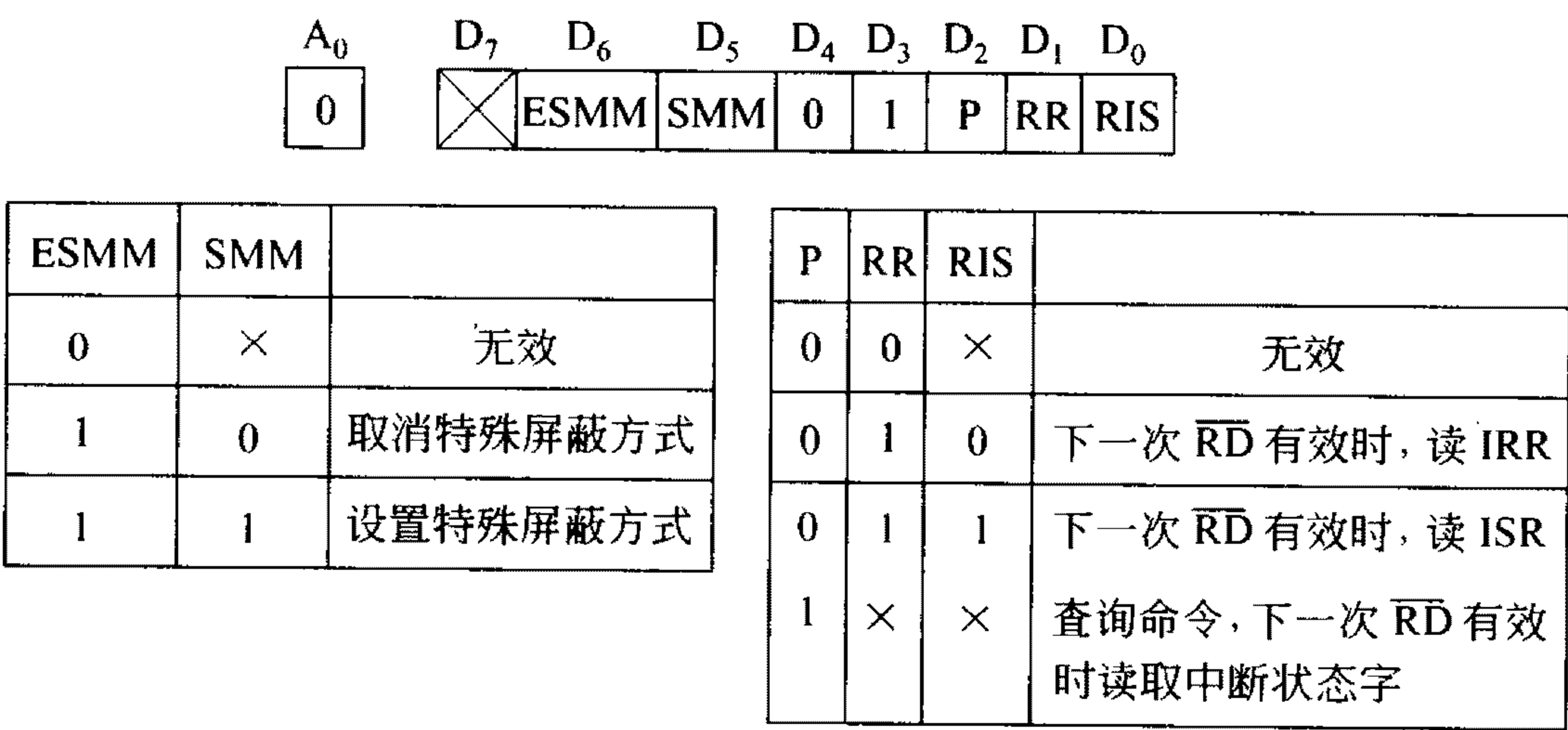


图 9-13 OCW₃ 格式

- ① OCW₃ 应写入偶地址端口, $D_4、D_3$ 位 = 01,是该命令字的标志,OCW₃ 有 3 个功能。
- ② ESMM、SMM=11,设置特殊屏蔽方式。ESMM、SMM=10 取消特殊屏蔽方式,恢复成常规屏蔽方式。
- ③ P 位是查询位,P 位=1 是一条查询命令。在这之后,针对偶地址端口执行一条读命令可得到 8259A 当前的中断状态字,中断状态格式如图 9-14 所示。
- 状态字 $D_7=1$ 表明当前有中断请求。此时, $D_2 \sim D_0$ 就是当前响应的中断源编码。 D_7 位=0 表示无中断请求。

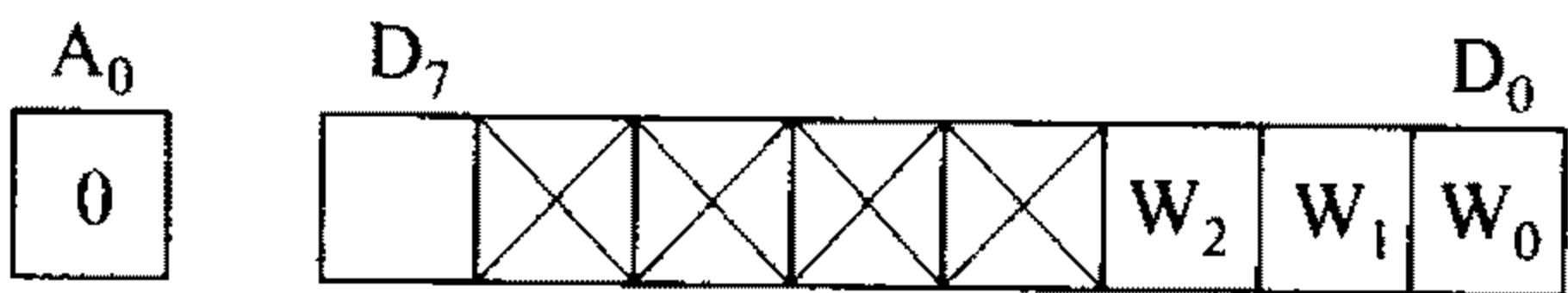


图 9-14 中断状态字

- ④ RR 位为寄存器读出控制位。当 RR 位=1 时,表示准备读取某个寄存器的内容。
- 具体说,当写入 OCW₃ 的 P 位=0,RR、RIS=10 时,下一次针对偶地址端口执行一条 IN 指令,读出的就是 IRR 的内容。
- 当 P 位=0,RR、RIS=11 时,下一次针对偶地址端口执行 IN 指令读出的就是 ISR 寄存器的内容。

初始化之后,任何时刻针对奇地址端口执行 IN 指令就可读出中断屏蔽寄存器 IMR 的内容,而不需要先设置读出命令。

3. 8259A 初始化编程步骤

8259A 工作之前要按照一定的顺序,将初始化命令字(不是操作命令字)ICW₁~ICW₄ 写入规定的端口。初始化编程之后,8259A 就可以工作了。在级连方式的主-从结构中,8259A 初始化应按顺序写入 ICW₁,ICW₂,ICW₃ 和 ICW₄。主、从 8259A 应分别初始化。

PC/AT 微型计算机以 80286 为 CPU,系统采用两片 8259A 管理 15 级可屏蔽中断,系统分配给主 8259A 的端口地址为 20H,21H,分配给从 8259A 的端口地址为 A0H,A1H,系统加电后由 BIOS 对主、从 8259A 进行初始化编程。

8259A 的初始化流程如图 9-15 所示。

对主 8259A 的初始化编程如下:

```

MOV    AL,11H    ;ICW1
OUT    20H,AL    ;预置中断请求为边沿触发
JMP    $+2       ;延时,转下条指令
MOV    AL,8      ;ICW2
OUT    21H,AL    ;预置主 8259A 中断类型码高 5 位为 00001
JMP    $+2
MOV    AL,4      ;主 ICW3
OUT    21H,AL    ;表示主 8259A 的 IR2 接有从 8259A
JMP    $+2
MOV    AL,1      ;ICW4
OUT    21H,AL    ;完全嵌套,非缓冲方式,非自动中断结束
JMP    $+2
.....

```

对从 8259A 的初始化编程如下:

```

MOV    AL,11H    ;ICW1
OUT    0A0H,AL   ;预置中断请求为边沿触发
JMP    $+2       ;延时,转下条指令
MOV    AL,70H    ;ICW2
OUT    0A1H,AL   ;预置从 8259A 中断类型码高 5 位为 01110
JMP    $+2
MOV    AL,2      ;ICW3
OUT    0A1H,AL   ;设置从 8259A 设备代码为 2
JMP    $+2
MOV    AL,1      ;ICW4

```

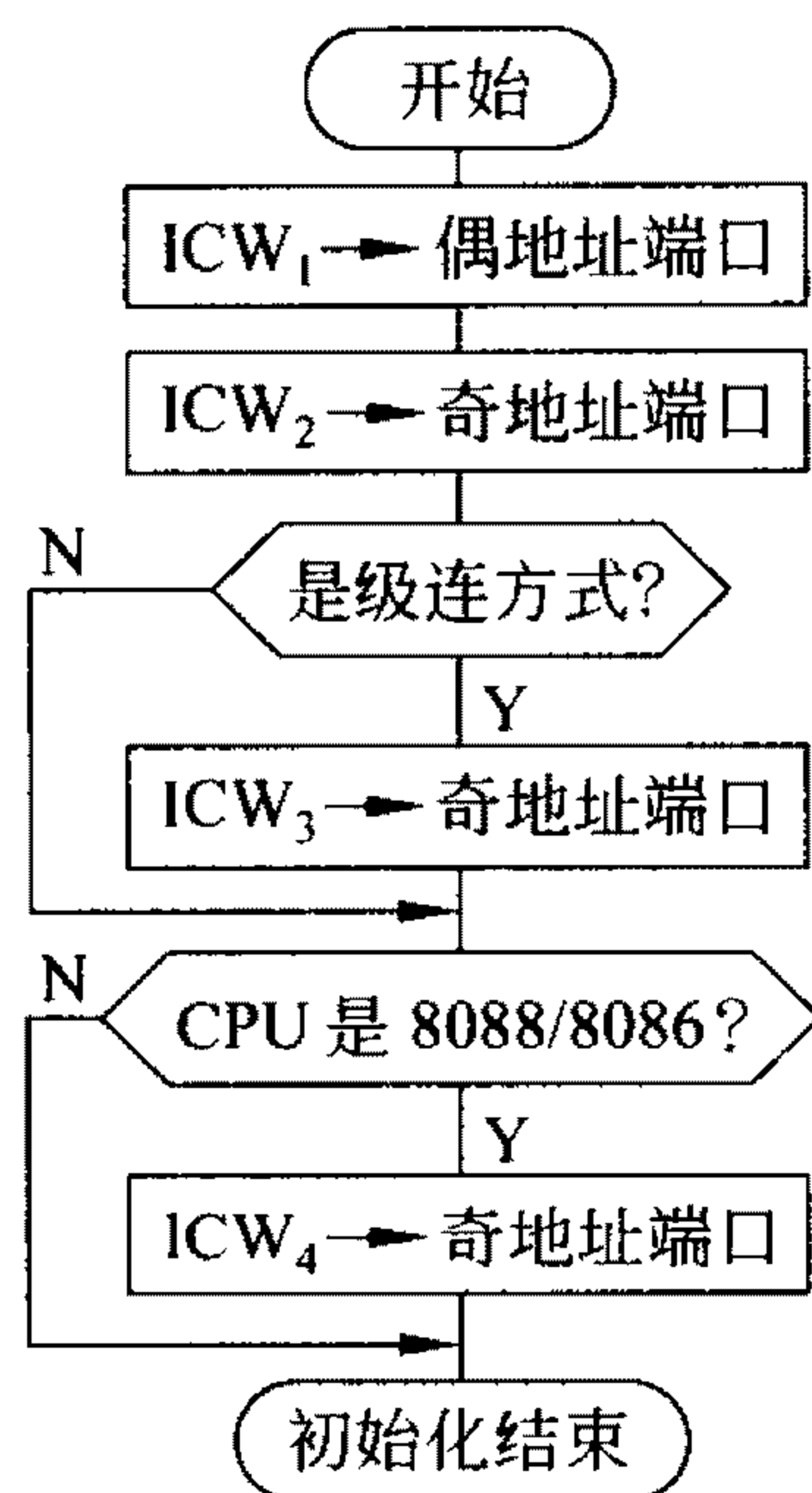


图 9-15 8259A 初始化流程

```

OUT    0A1H,AL    ;完全嵌套,非缓冲方式,非自动中断结束
JMP     $ + 2
.....

```

4. 敬告读者

8259A 中断控制器是中断系统的核心器件,对它的初始化编程要涉及中断系统的软硬件许多问题,而且一旦完成初始化,所有硬件中断源和中断处理程序(包括已开发和未开发的)都必须受其制约。这是一项影响深远而且带有决策性质的问题,只有微型计算机系统的设计者才有权定夺。正因为如此,对系统 8259A 的初始化编程是在微型计算机启动之后由 BIOS 自动完成的。从系统安全性考虑,用户在使用过程中不应当再对其初始化,更不能改变对它的初始化设置。

我们的学习重点是掌握系统对 8259A 初始化之后,确定了哪些中断管理方式。使用者只有严格遵守这些约定,才能开发新的中断。阅读了 BIOS 对系统 8259A 的初始化程序,我们归纳出以下几点:

① 系统 8259A 的中断触发方式采用边沿触发,当我们开发新的中断源的时候。中断请求信号必须是从低电平向高电平的跃变。

② 中断屏蔽采用常规屏蔽方式。在开发中断处理程序的时候,可以随时向中断屏蔽寄存器(主 8259A 端口地址为 21H,从 8259A 端口地址为 A1H)写入新的中断屏蔽字以便开放/禁止某一级中断。

③ 系统 8259A 中断优先级管理采用完全嵌套即固定优先级方式,IR₀ 的中断级别最高……IR₇ 的中断级别最低。

④ 中断结束,采用常规结束方式。这就是讲在设计硬件中断(不是软件中断)服务程序的时候,在中断处理结束,中断返回之前,必须向 8259A 的偶地址端口(主 8259A 端口地址为 20H,从 8259A 端口地址为 A0H)传送一个中断结束命令,以便使 8259A 中断服务寄存器中的 ISR_i 位清零,通知 8259A 本次中断结束。常用的中断结束命令字(OCW₂)为 20H。

9.6 微型计算机系统可屏蔽中断

9.6.1 可屏蔽中断与非屏蔽中断

80x86 有 2 个引脚(INTR 和 NMI)可以接收外部的硬件中断请求,INTR 引脚上的中断请求引发的中断称为可屏蔽中断,NMI 引脚上的中断请求引发的中断称为非屏蔽中断,可屏蔽中断、非屏蔽中断都是硬件中断。非屏蔽中断的级别高于可屏蔽中断,DMA 请求的级别比非屏蔽中断的级别更高。

CPU 响应可屏蔽中断的条件是:

- ① INTR 引脚有中断请求,NMI 引脚没有中断请求,系统没有 DMA 请求。
- ② CPU 当前指令执行完毕。

③ CPU 处于开中断状态,即标志寄存器的中断允许标志置 1。

CPU 响应非屏蔽中断的条件是:

- ① NMI 引脚有中断请求,系统没有 DMA 请求。
- ② CPU 当前指令执行完毕。

CPU 在每一条指令的最后一个时钟周期,检测 INTR 和 NMI 引脚,如果有非屏蔽中断请求,在满足上述条件之后,CPU 自动转向 2 型服务程序,处理非屏蔽中断。当检测到有可屏蔽中断请求时,在满足上述条件的前提下,通过总线控制器向系统 8259A 发出中断响应信号(2 个负脉冲)。在获得 8259A 送来的中断类型码之后,在实地址模式下查询中断向量表,从而转向相应中断源的中断服务程序。

9.6.2 可屏蔽中断的硬件结构

1. 结构分析

图 9-16 画出了 80286 以上微型计算机系统的可屏蔽中断硬件结构。系统的可屏蔽中断使用两片 8259A 管理 15 级中断。系统分配给主 8259A 的端口地址为 20H 和 21H,分配给从 8259A 的端口地址为 A0H 和 A1H。

系统初始化以后,主从 8259A 均按“固定优先级”方式管理它属下的中断源,当从 8259A 任一中断源请求被选中后,经由从 8259A 的 INT 端向主 8259A 的 IR₂ 提出请求,因此整个系统中断源的级别从高到低依次为:主 IR₀、IR₁,从 IR₀~IR₇,主 IR₃~IR₇。

按照 BIOS 初始化的规定,主从 8259A 均采用常规方式结束中断,这就是讲,主 8259A 属下的中断源其中断服务程序结束,执行 IRET 指令之前,应向主 8259A 20H 端口地址写一个常规中断结束命令字,即执行以下两条指令:

```
MOV    AL,20H
OUT    20H,AL
```

同理,从 8259A 属下的中断源,在结束中断服务之前应执行下列指令,通知从 8259A 本次中断结束:

```
MOV    AL,20H
OUT    0A0H,AL
```

各级中断源及其中断类型码的分配如表 9-1 所示。

表 9-1 硬件中断源与中断类型对照表

主 8259A	中断源	中断类型	从 8259A	中断源	中断类型
IR ₀	日时钟	08H	IR ₀	实时时钟	70H
IR ₁	键盘	09H	IR ₁	用户中断	71H 改向 0AH
IR ₂	来自从 8259A		IR ₂	保留	72H
IR ₃	辅串口	0BH	IR ₃	保留	73H

续表

主 8259A	中断源	中断类型	从 8259A	中断源	中断类型
IR ₄	主串口	0CH	IR ₄	保留	74H
IR ₅	并行口 2	0DH	IR ₅	协处理器	75H
IR ₆	软盘	0EH	IR ₆	硬盘	76H
IR ₇	并行口 1	0FH	IR ₇	保留	77H

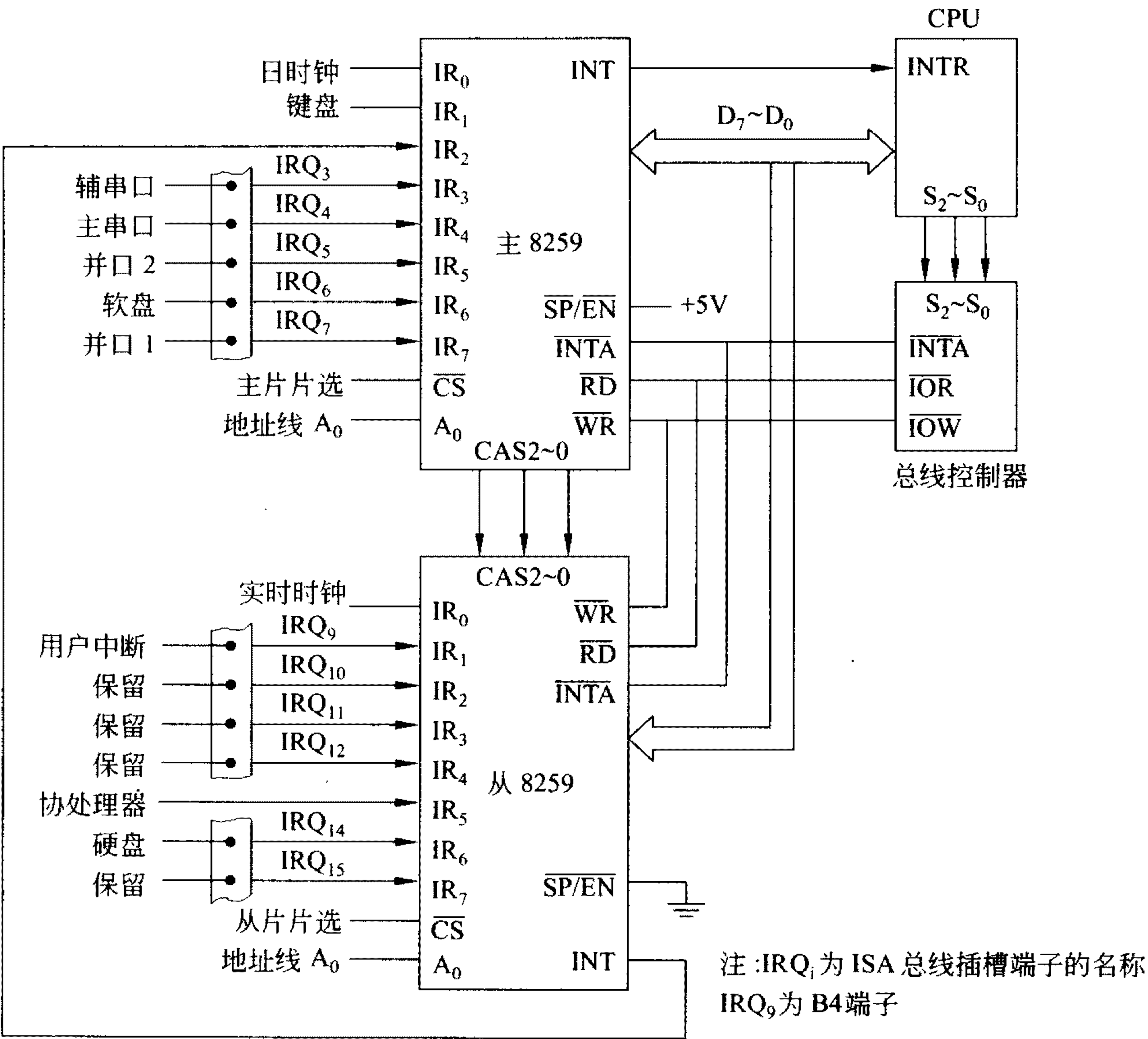


图 9-16 可屏蔽中断硬件结构

2. 用户中断

用户中断是微型计算机系统为用户开发可屏蔽中断预备的中断口,用户中断的过程如下:

用户中断请求,通过系统的 ISA 总线 B4 端子引入,接到从 8259A 的 IR₁,经过从 8259A 中断屏蔽寄存器 IMR D₁ 位的屏蔽/开放管理和优先级判优,再经过 INT 端接至主 8259A 的 IR₂,经过主 8259A 中断屏蔽寄存器 IMR D₂ 位的屏蔽/开放管理和主 8259A 的优先权电路,最终被主 8259A 选中,由主 8259A 向 CPU 提出中断请求,CPU 响应用户中断后,从 8259A 送出的中断类型码是 71H,于是 CPU 转向 71H 型服务程序。BIOS 设

计的 71H 型服务程序如下：

```
PUSH    AX
MOV     AL,20H
OUT     0A0H,AL
POP     AX
INT     0AH
```

该程序做两件事：向从 8259A 发出中断结束命令，然后主动执行 INT 0AH，转向 0AH 服务程序。这就是讲，最终的用户中断类型为 0AH 型，用户中断的服务程序应预先设计好，并且其入口地址应放入系统 RAM $4 \times 0AH \sim 4 \times 0AH + 3$ 单元。为什么用户中断要经过这样曲折的途径呢？这完全是为了和早期的 PC/XT 机型兼容而设置的。

我们提请读者注意一个问题，在国内微型计算机市场上，除了原装机之外，还有众多的品牌机、兼容机。仅仅就用户中断涉及的屏蔽位（主 8259A IMR D_2 位，从 8259A IMR D_1 位）而言，其初始化设置就不尽相同，这给我们开发用户中断增添了麻烦，不过问题一经曝光总会有解决的办法。

我们建议在开发用户中断的时候做到以下几点：

(1) 硬件方面

把外扩中断源的中断请求（由低电平到高电平的跃变）接入 ISA 总线 B4 端子。

(2) 软件方面

① 置换 0AH 型中断向量，即调用 DOS 系统 25H 号子程序把用户中断的服务程序入口地址写入 $4 \times 0AH \sim 4 \times 0AH + 3$ 单元。

② 不论机型如何，分别向主从 8259A 写入中断屏蔽字，使主 8259A 的 IMR 的 D_2 位置 0，即开放从 8259A 中断，使从 8259A 的 IMR 的 D_1 位置 0，开放用户中断。

③ 每一次中断服务结束，执行 IRET 之前向主 8259A 送中断结束命令，通报本次中断结束。

④ 中断程序（不是中断服务程序）结束，返回 DOS 之前，分别向主从 8259A 写入中断屏蔽字，使主 8259A 的 IMR 的 D_2 位置 1，屏蔽从 8259A 中断，使从 8259A 的 IMR 的 D_1 位置 1，屏蔽用户中断。

9.6.3 硬件中断和软件中断的区别

综合以上描述可以看出，CPU 在获得中断类型码以后，如何转向服务程序，就这一点而言，响应硬件中断和软件中断的操作是相同的，但是硬件中断和软件中断有许多不同点：

① 中断的引发方式不同：硬件中断是由 CPU 以外的硬件设备发出中断请求（接到引脚 INTR 和 NMI）而引发的。而软件中断是由于 CPU 执行 INT n 指令而引发的。

② CPU 获取中断类型码的方式不同：响应硬件可屏蔽中断后，中断类型码是由 8259A 提供的。响应软件中断时，中断类型码是由软件中断指令 INT n 本身提供的。

③ CPU 响应的条件不同：CPU 只有在开中断时，才能响应硬件可屏蔽中断，相应软

件中断不受此限制。

④ 中断处理程序的结束方式不同。

在硬件可屏蔽中断服务程序中,中断处理结束后,需要做两件事:

一是向 8259A 发出中断结束命令,8259A 收到此命令后将 ISR 寄存器中的相应位清零,结束中断。二是执行 IRET 指令,中断返回。

而在软件中断服务程序中,中断处理结束后只需执行 IRET 指令。这些都是设计中中断服务程序必须掌握的基本概念。

9.7 日时钟中断

日时钟中断的中断源为系统 8254 的 0 号计数器,该计数器也由 BIOS 初始化,初始化以后,每隔 55ms 向主 8259A 的 IR_0 端子提请一次中断,CPU 响应后,转入 8 型中断服务程序,即日时钟中断处理程序。

1. 日时钟中断处理流程

BIOS 设计的中断处理流程,依次完成如下操作:

- ① 开中断,保护现场,40H→DS。
- ② 对“日时钟计数器”进行一次加 1 计数。
- ③ 测算软驱马达的关闭时间。
- ④ 执行 INT 1CH 指令。
- ⑤ 向主 8259A 发出常规中断结束命令。
- ⑥ 恢复现场,执行 IRET 指令。

什么是“日时钟计数器”?

BIOS 规定:系统 RAM 40H: 6CH~40H: 6FH 这 4 个单元为“日时钟计数器”。每响应一次日时钟中断,对该计数器加 1,计满 001800B0H 即为 24 小时。满 24 小时后,BIOS 自动使计数单元清零,准备新的计数,并且使 40H: 70H 单元为 1,表示新的一天开始。

日时钟计数器中的计数值,可以通过 INT 1AH 的 0 号功能调用读取,1 号功能调用预置。但需注意,读取和预置的值都是计数值而不是时间值,因此用户程序很少使用。

对软盘进行读写操作之后,软驱马达应及时关闭。日时钟中断处理程序负责测算软驱马达的关闭时间。

做完以上两件事之后,日时钟中断处理程序主动执行一条“INT 1CH”指令,而 BIOS 为 1CH 型中断设计的服务程序只有一条 IRET 指令,故而立即返回,结束日时钟中断。

2. 日时钟中断的外扩

上面讲到,日时钟中断每次都要执行 INT 1CH,这就是讲,系统正常工作之后,每隔 55ms 都要访问一次 1CH 型中断,而 BIOS 为 1CH 型中断只设计了一条 IRET 指令,这

实际上是 BIOS 为用户预留的“缺口”。如果用户有一项定时操作,只要该操作的定时周期为 55ms 的整数倍,就可以为此设计一个定时操作程序,并用它取代 1CH 型中断。通常,我们把 1CH 型中断称为日时钟的外扩中断。

9.8 实地址模式定时中断程序设计

9.8.1 定时中断程序的设计方法

设计定时中断程序必须考虑服务程序中断类型的选择、中断向量的置换、用户设计的定时中断服务程序与系统服务程序之间的衔接和避免 DOS 重入等问题。对这些问题的考虑,也适用于设计其他中断程序。

1. 中断类型的选择

以定时中断为例,我们知道系统的 8 型中断是日时钟中断,1CH 型中断又是外扩的日时钟中断。8 型、1CH 型都是每隔 55ms 一次的定时中断。系统定时器 8254 的 0 通道是系统定时中断的中断源。

在设计定时中断程序的时候,首先要明确一个问题,谁是定时源? 定时源不同,定时中断服务程序的类型也不一样,定时源有两种可能的选择。

(1) 采用外扩定时源

例如:用可编程定时器,或者不可编程的器件自制一个定时电路,这就是外扩定时源。外扩定时脉冲(由低电平到高电平的跃变)应当接到系统 ISA 总线的 B4 端子。

采用外扩定时源,定时中断服务程序的类型为 0AH 型,即做为用户中断处理。

(2) 借用系统定时源

借用系统定时源完成定时操作,如何选择中断类型呢?

① 当用户程序的某项定时操作,其定时周期等于 55ms 的整数倍时,可定义用户程序的定时操作为 1CH 中断,并为此设计一个中断服务子程序,取代系统原先的 1CH 型中断服务程序。在新的 1CH 型服务程序中,对日时钟中断进行“中断计数”,计满 N 次执行一次预定的操作, $N \times 55\text{ms}$ 是用户定时操作程序的定时周期。

② 当用户程序的定时操作,其定时周期不等于 55ms 的整数倍,或者小于 55ms 时,必须采取下列措施:

* 改变系统定时器的定时时间

由用户程序对系统定时器重新初始化,使之提请中断的时间等于用户程序定时操作的周期和 55ms 之间的最大公约数 X。

- 定义定时操作中断把用户程序的定时操作定义为 08H 型中断,取代系统的日时钟中断。

- 完成新旧 08H 型中断服务程序之间的衔接

由于系统定时器的定时时间改变了(现在是每隔 Xms 有一次中断请求),为了不破坏“日计时”的正确性,在新的 08H 型中断服务程序中需要对定时中断进行计数,达

到定时操作时间时,进行定时操作,满 55ms 时,执行一次系统原有的日时钟中断服务程序。

2. 中断向量的置换

服务程序的中断类型(假设为 N 型)确定之后,接下来要考虑的是:一旦发生中断怎样引导 CPU 执行用户的(而不是系统的)服务程序呢?这就需要预先进行中断向量的置换。用 DOS 系统 35H 号子程序,将中断向量表中原有的 N 型中断向量读出并保存到用户程序的数据区,再用 DOS 系统的 25H 号子程序将用户服务程序的中断向量写入系统 RAM $4 \times N \sim 4 \times N + 3$ 单元。程序结束(不是中断服务程序结束!)返回 DOS 之前再恢复系统中断向量的设置。中断向量的置换、中断向量的恢复,这是设计中断程序的重要措施。

3. 避免“DOS 重入”

“DOS 不可重入”是很复杂的问题,深入讲述 DOS 不可重入的机理已超出本章的范畴。但是在设计中断程序和驻留程序的时候,不可避免地要涉及 DOS 重入的问题,本小节只作简单介绍。

中断是随机发生的,被中断的程序称为现行程序或者简称主程序,中断发生后,CPU 转向的是中断服务程序。

什么是“DOS 重入”呢?简单地说,当主程序正在执行 INT 21H 的某项子功能时,该功能调用还没有结束,X 中断源提出了中断请求,CPU 响应后,中断该项子功能的执行,从 21H 功能退出,转而执行 X 中断服务程序,如果 X 中断服务程序又要执行 INT 21H 指令,则 CPU 又要“重新进入”DOS,这一过程称为“DOS 重入”,DOS 不允许重入!强行使 DOS 重入很可能使系统瘫痪。

避免 DOS 重入最简单的方法是服务程序中不调用 INT 21H 功能,或者主程序、服务程序不同时调用 INT 21H。在服务程序中若要进行 I/O 操作、屏幕显示,可调用相应的 BIOS 功能。

4. 中断服务程序的执行时间

定时中断服务程序的执行时间,必须远远小于定时中断的时间间隔。

5. 中断服务程序的返回

在完成“中断服务”之后,中断服务程序必须向 8259A 发送中断结束命令,通报本次中断结束,然后才能执行 IRET 指令。否则 8259A 中断服务寄存器中相应的 ISR 位不能复位,8259A 不能受理同一中断源的再次中断请求。

9.8.2 定时中断程序设计举例

【例 9.8.1】1CH 型中断的应用

假设微型计算机系统外扩了如图 9-17 所示的数码管电路,要求使用系统定时源并采

用中断方式,每隔一秒完成一次8字左移,循环往复,直到主机键盘按下任意键时停止。

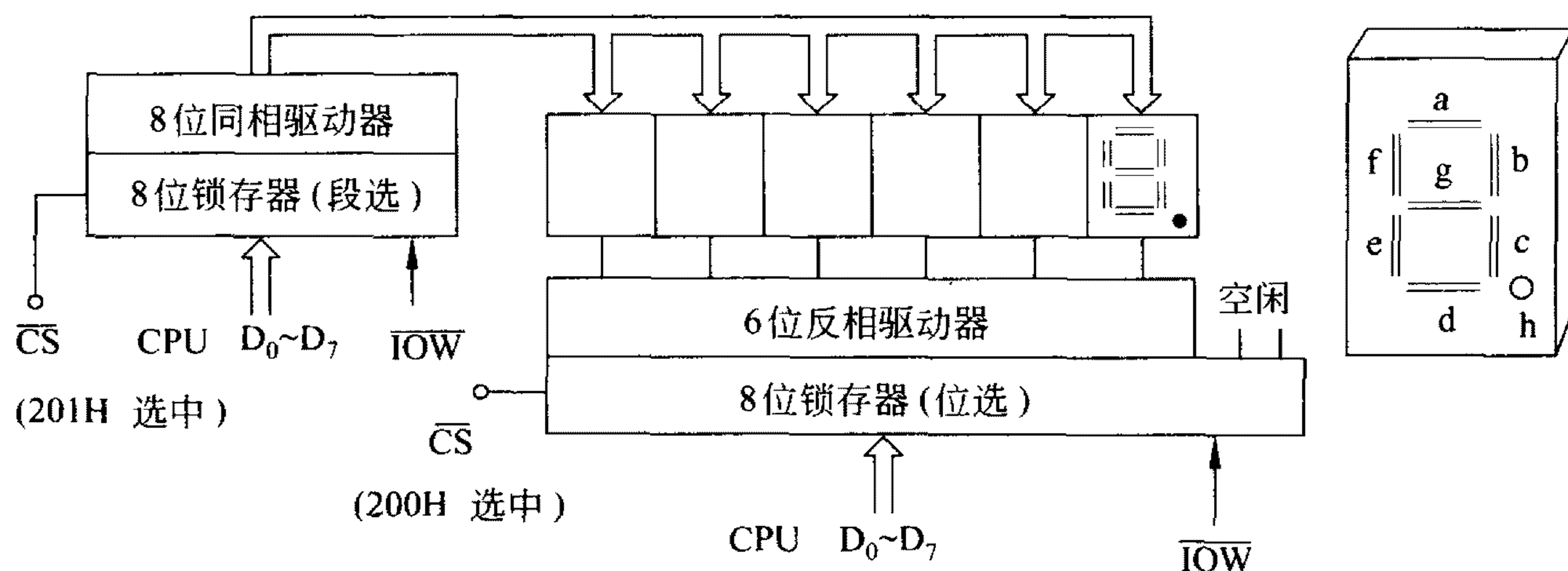


图 9-17 外扩数码管显示电路

【电路分析】

图中数码管为共阴极,6个数码管的同名段已复接,该电路有两个端口,即段选端口201H,位选端口200H。

针对端口地址201H执行输出指令时,段选寄存器锁存数据线 $D_0 \sim D_7$ 的信息,经过同相驱动器,驱动6个数码管的a~h阳极段。即数据线 D_0 对应于a段,……,数据线 D_7 对应于h段。

针对端口地址200H执行输出指令时,位选寄存器锁存数据线 $D_0 \sim D_7$ 位的信息,经过反相驱动器,驱动第1位~第6位数码管的共阴极。当 D_0 位为1时,点亮第1位数码管, D_0 位为0时,熄灭第1位数码管,以此类推。

【设计思路】

因为系统定时器每隔55ms产生一次日时钟中断,中断18次为990ms,接近一秒钟。还由于日时钟中断每次都要调用1CH型中断,因此用户可以设计新的1CH型中断服务程序,对日时钟中断进行计数,计满18次完成一次8字左移。

对日时钟中断进行“中断计数”,是实现长时间定时的常用方法。程序框图如图9-18所示。

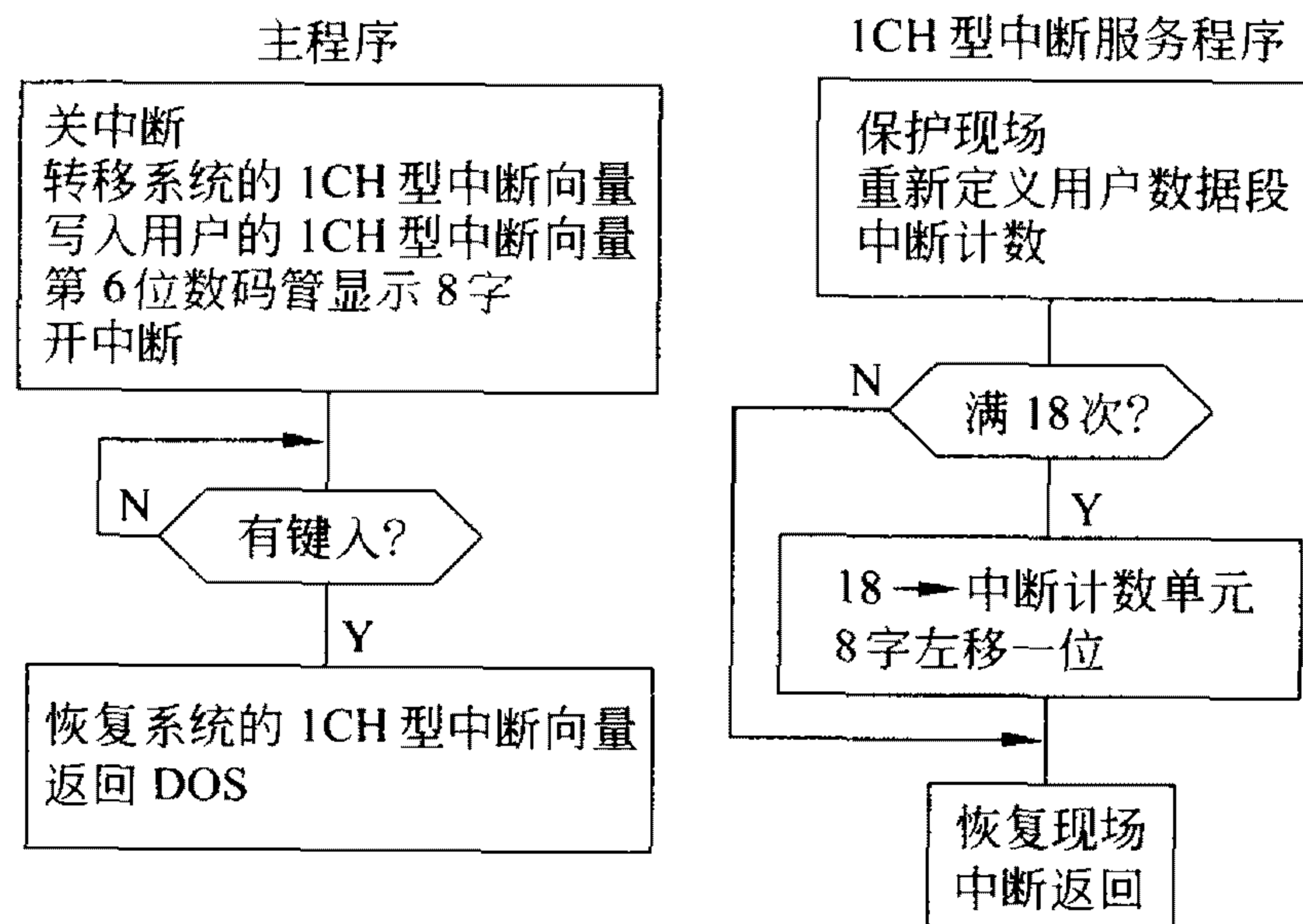


图 9-18 【例 9.8.1】程序框图

【程序清单】

```

;FILENAME: 981. ASM
.486
DATA SEGMENT USE16
OLD1C DD ?
ICOUNT DB 18 ;中断计数初值
ORIGIN DB 20H ;数码管位选初值
DATA ENDS
CODE SEGMENT USE16
ASSUME CS: CODE,DS: DATA
BEG: MOV AX,DATA
MOV DS,AX
CLI ;关中断
CALL READ1C
CALL WRITE1C
MOV DX,201H
MOV AL,7FH
OUT DX,AL ;输出 8 的字型码
MOV DX,200H
MOV AL,ORIGIN
OUT DX,AL ;定位显示
STI ;开中断
SCAN: MOV AH,1
INT 16H ;有键入?
JZ SCAN ;否转
CALL RESET
MOV AH,4CH
INT 21H
;-----
SERVICE PROC
PUSHA ;保护现场
PUSH DS ;DS=40H
MOV AX,DATA
MOV DS,AX ;重新给 DS 赋值
DEC ICOUNT ;中断计数
JNZ EXIT ;不满 18 次转移
MOV ICOUNT,18 ;满 18 次重新设置计数初值
SHR ORIGIN,1
JNC NEXT
MOV ORIGIN,20H
NEXT: MOV AL,ORIGIN
MOV DX,200H
OUT DX,AL ;8 字左移一位
EXIT: POP DS ;恢复现场

```

```

                POPA
                IRET                                ;返回系统 8 型中断服务程序
SERVICE       ENDP
;-----
READ1C         PROC                                ;转移系统 1CH 型中断向量
                MOV     AX,351CH
                INT     21H
                MOV     WORD PTR OLD1C,BX
                MOV     WORD PTR OLD1C+2,ES
                RET
READ1C         ENDP
;-----
WRITE1C        PROC                                ;写入用户 1CH 型中断向量
                PUSH    DS
                MOV     AX,CODE
                MOV     DS,AX
                MOV     DX,OFFSET SERVICE
                MOV     AX,251CH
                INT     21H
                POP     DS
                RET
WRITE1C        ENDP
;-----
RESET          PROC                                ;恢复系统 1CH 型中断向量
                MOV     DX,WORD PTR OLD1C
                MOV     DS,WORD PTR OLD1C+2
                MOV     AX,251CH
                INT     21H
                RET
RESET          ENDP
CODE           ENDS
END            BEG

```

【程序分析】

① 程序中 READ1C 子程序读出系统的 1CH 型中断向量,并转移到用户数据段的 OLD1C 双字单元,准备在恢复 1CH 型中断向量时使用。

WRITE1C 子程序把用户的服务程序中断向量写入中断向量表的 $4 \times 1\text{CH} \sim 4 \times 1\text{CH} + 3$ 单元中,完成 1CH 型中断向量的置换,这些操作应当在关中断的条件下进行。

② 程序中使用 INT 16H 的 1 号功能,查询键盘缓冲区,并且等待定时中断。

③ 进入服务程序(即 SERVICE 子程序)之后,为什么要重新定义用户数据段呢?这是因为服务程序中,需要对用户数据段进行操作。

当日时钟中断发生时,系统转入日时钟中断处理程序,该程序首先把被中断程序的 DS 值压栈,然后设置 $\text{DS} = 0040\text{H}$ (因为日时钟计数器在 $40\text{H}:6\text{CH} \sim 40\text{H}:6\text{FH}$)进行一

次日计时,……,之后才能执行 INT 1CH。也就是说,在转入 1CH 型服务程序(现在被用户的 SERVICE 服务子程序取代了)时,DS 值已经不再是用户程序数据段的段基址了,如果服务程序需要访问用户数据段单元,当然要重新设置 DS 的值。

④ RESET 子程序恢复系统的 1CH 型中断向量。使用 INT 21H 的 25H 号子功能需要注意的是,在设置入口参数时,先设置 DX 的值,后设置 DS 的值,二者顺序不可颠倒。

【例 9.8.2】 8 型中断的应用。

假设系统外扩了图 9-17 所示的数码管电路,仍然要求在定时中断的条件下,每隔 1 秒钟完成一次 8 字左移,直到主机键盘按下任意键时停止。

【设计思路】

如果 8 字左移这一定时操作改用 08H 型中断完成,程序应采取下列措施:

① 重新对系统定时器初始化,使之每隔 5ms 提请一次中断(5ms 是 1 秒和 55ms 的最大公约数)。

② 定义用户的定时操作为 08H 型中断,取代系统的日时钟中断服务程序。

③ 在新的 08H 型中断服务程序中进行中断计数,计满 200 次(正好 1 秒钟)完成一次 8 字左移。计满 11 次(正好 55ms)调用一次日时钟服务程序,完成新旧 08H 型中断服务程序之间的衔接。程序框图如图 9-19 所示。

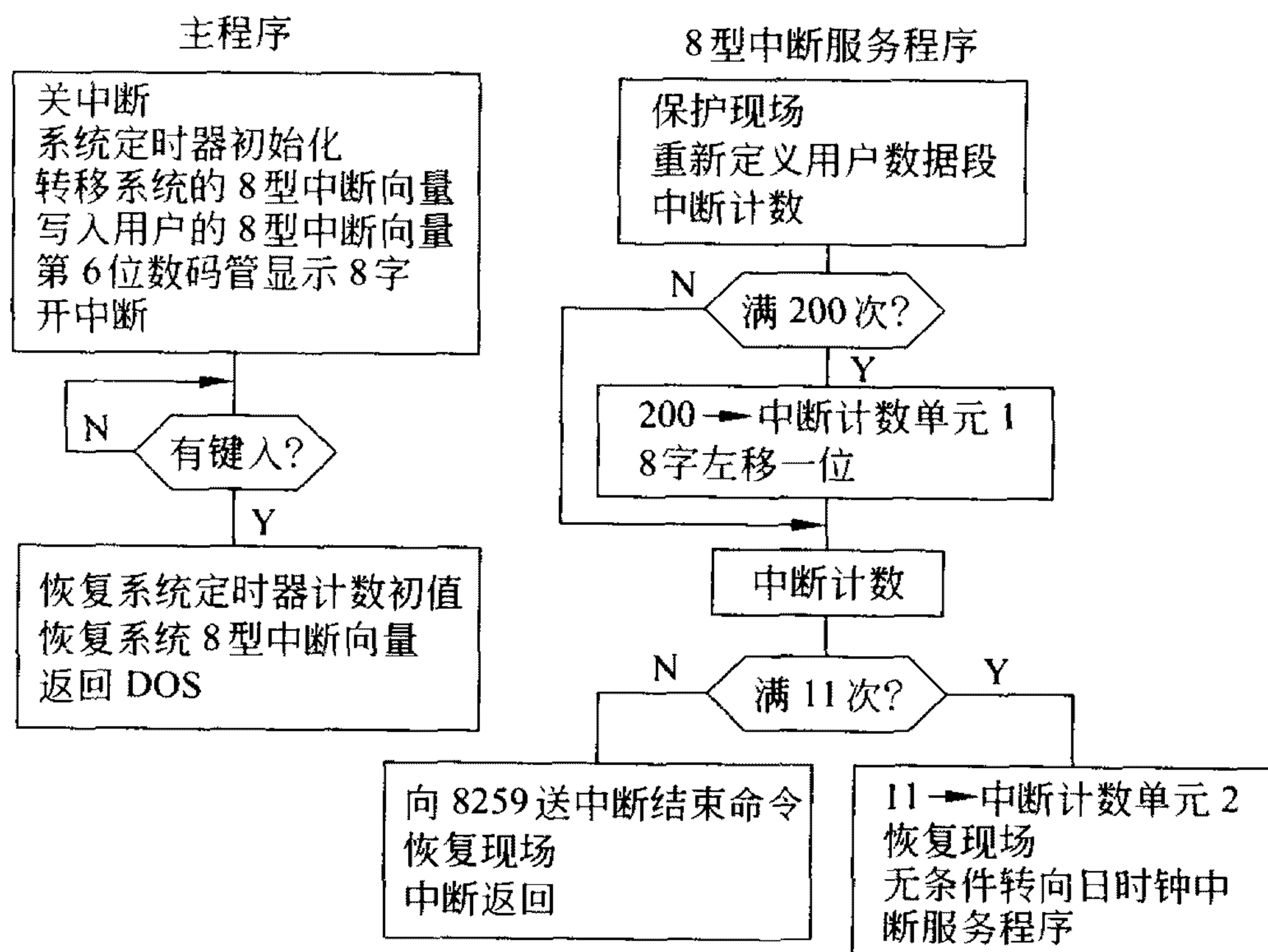


图 9-19 【例 9.8.2】程序框图

【程序清单】

```

;FILENAME: 982. ASM
.486
DATA SEGMENT USE16
COUNT1 DB 200 ;秒中断计数初值
COUNT2 DB 11 ;55ms 中断计数初值
    
```



```

ORIGIN      DB      20H          ;数码管位选初值
DATA        ENDS
CODE        SEGMENT USE16
OLD08       DD      ?           ;存放系统 8 型中断向量
          ASSUME  CS: CODE,DS: DATA
BEG:        MOV     AX,DATA
          MOV     DS,AX
          CLI          ;关中断
          CALL    I8254
          CALL    READ08
          CALL    WRITE08
          MOV     DX,201H
          MOV     AL,7FH
          OUT     DX,AL          ;输出 8 的字型码
          MOV     DX,200H
          MOV     AL,ORIGIN
          OUT     DX,AL          ;定位显示
          STI          ;开中断
SCAN:       MOV     AH,1
          INT     16H          ;有键入?
          JZ      SCAN          ;无,转移
          CALL    RESET
          MOV     AH,4CH
          INT     21H
;-----
SERVICE    PROC
          PUSH    A          ;保护现场
          PUSH    DS          ;DS 可能是用户数据段段基址
          MOV     AX,DATA      ;也可能是 40H
          MOV     DS,AX        ;重新给 DS 赋值
          DEC     COUNT1       ;中断计数
          JNZ     NEXT2        ;不满 1 秒钟转移
          MOV     COUNT1,200    ;重新设置计数初值
          SHR     ORIGIN,1
          JNC     NEXT1
          MOV     ORIGIN,20H
NEXT1:      MOV     AL,ORIGIN
          MOV     DX,200H
          OUT     DX,AL        ;8 字左移一位
NEXT2:      DEC     COUNT2      ;中断计数
          JNZ     EXIT         ;不满 55ms 转移
          MOV     COUNT2,11     ;重新设置计数初值
          POP     DS           ;恢复现场
          POPA          ;恢复现场

```

	JMP	CS: OLD08	;转系统 8 型中断服务程序
EXIT:	MOV	AL,20H	;中断结束命令
	OUT	20H,AL	;→主 8259
	POP	DS	;恢复现场
	POPA		;恢复现场
	IRET		;返回被中断程序
SERVICE	ENDP		
;			
I8254	PROC		;系统定时器初始化
	MOV	AL,36H	
	OUT	43H,AL	
	MOV	AX,5965	;每隔 5ms
	OUT	40H,AL	
	MOV	AL,AH	
	OUT	40H,AL	;提一次中断请求
	RET		
I8254	ENDP		
;			
READ08	PROC		;转移系统 8 型中断向量
	MOV	AX,3508H	
	INT	21H	
	MOV	WORD PTR OLD08,BX	
	MOV	WORD PTR OLD08+2,ES	
	RET		
READ08	ENDP		
;			
WRITE08	PROC		;写入用户 8 型中断向量
	PUSH	DS	
	MOV	AX,CODE	
	MOV	DS,AX	
	MOV	DX,OFFSET SERVICE	
	MOV	AX,2508H	
	INT	21H	
	POP	DS	
	RET		
WRITE08	ENDP		
;			
RESET	PROC		;恢复系统资源
	MOV	AL,0	
	OUT	40H,AL	
	OUT	40H,AL	
	MOV	DX,WORD PTR OLD08	
	MOV	DS,WORD PTR OLD08+2	
	MOV	AX,2508H	

```

                INT      21H
                RET
RESET          ENDP
CODE           ENDS
                END      BEG

```

【程序分析】

① I8254 子程序对系统定时器初始化,使之每隔 5ms 提请一次中断。

② READ08 子程序读取日时钟中断的中断向量,并存入 OLD08 双字单元。注意,中断向量的存放规律必须是: OLD08~OLD08+1 单元存放中断向量的偏移地址, OLD08+2~OLD08+3 单元存放中断向量的段基址。

③ WRITE08 子程序,负责把新的中断服务子程序(过程名为 SERVICE)的入口地址写到系统 RAM $4 \times 08 \sim 4 \times 08 + 3$ 单元,从而取代系统的 8 型中断向量。

④ 为什么在 SERVICE 中断服务子程序中要重新给 DS 赋值,使其指向用户程序数据段呢?关键在于进入 SERVICE 子程序之后,DS 的值可能不是用户数据段的段基址,而 SERVICE 子程序必须对用户数据段进行操作。

用户程序在“开中断”之后,执行“MOV AH,1”,“INT 16H”和“JZ SCAN”3 条指令,这一小段程序有两个作用,其一,等待中断,其二,当有任意键键入时结束演示。

如果有中断发生,而且 CPU 是在这 3 条指令中的任何一条指令结束后响应中断(转入 SERVICE 子程序),则 DS 的值仍然等于用户数据段段基址,此时 SERVICE 子程序没有必要重新给 DS 赋值。

问题在于:当 CPU 执行“INT 16H”,转入 16H 型中断服务程序之后,BIOS 立即把被中断程序的 DS 值(对本例而言,DS 的值就是用户数据段段基址)压栈,再设置 DS 的值为 40H,使其等于键盘缓冲区的段基址,然后根据 AH 的值转入不同的程序段……。很显然,如果是在进入“键盘中断服务程序”之后,响应定时中断(转入 SERVICE 子程序),则 DS 的值绝不是用户数据段的段基址,而 SERVICE 子程序恰恰要对用户程序数据段进行操作,为了保证寻址的正确性,SERVICE 子程序当然要对 DS 重新赋值,使其等于用户数据段的段基址。

⑤ 本程序改变了系统定时源的计数初值,又用 SERVICE 子程序取代了系统的 8 型中断服务程序,为了不破坏系统资源,保证日时钟计数的正确性,采取了两项措施:

每响应一次中断,SERVICE 子程序分别对 COUNT1,COUNT2 单元减 1 计数。对 COUNT2 单元中断计数 11 次之后(即为 55ms),执行“JMP CS: OLD08”段间转移指令,转入日时钟服务程序。对 COUNT1 单元中断计数 200 次之后(即为 1 秒钟),完成一次 8 字左移,并且向 8259A 发出中断结束命令,之后再中断返回。

OLD08 双字单元必须定义在代码段,不能设置在数据段,这是最关键的。从程序中可以看出,在执行“JMP CS: OLD08”之前,必须完成“恢复现场”的操作,恢复现场之后,DS 的值可能是用户数据段的段基址,也可能是“键盘缓冲区”的段基址 40H,如果 OLD08 双字单元设置在数据段,恢复现场之后 DS 的值又是 40H,则执行“JMP OLD08”之后系统将死机。

9.9 实时时钟中断

9.9.1 实时时钟电路

微型计算机系统的主板上配置了一个实时时钟电路,该电路以一片 RT/CMOS RAM 为核心,内部有 64 个字节存储单元(PC/AT 机),支持实时时钟,图 9-20 为其示意图。

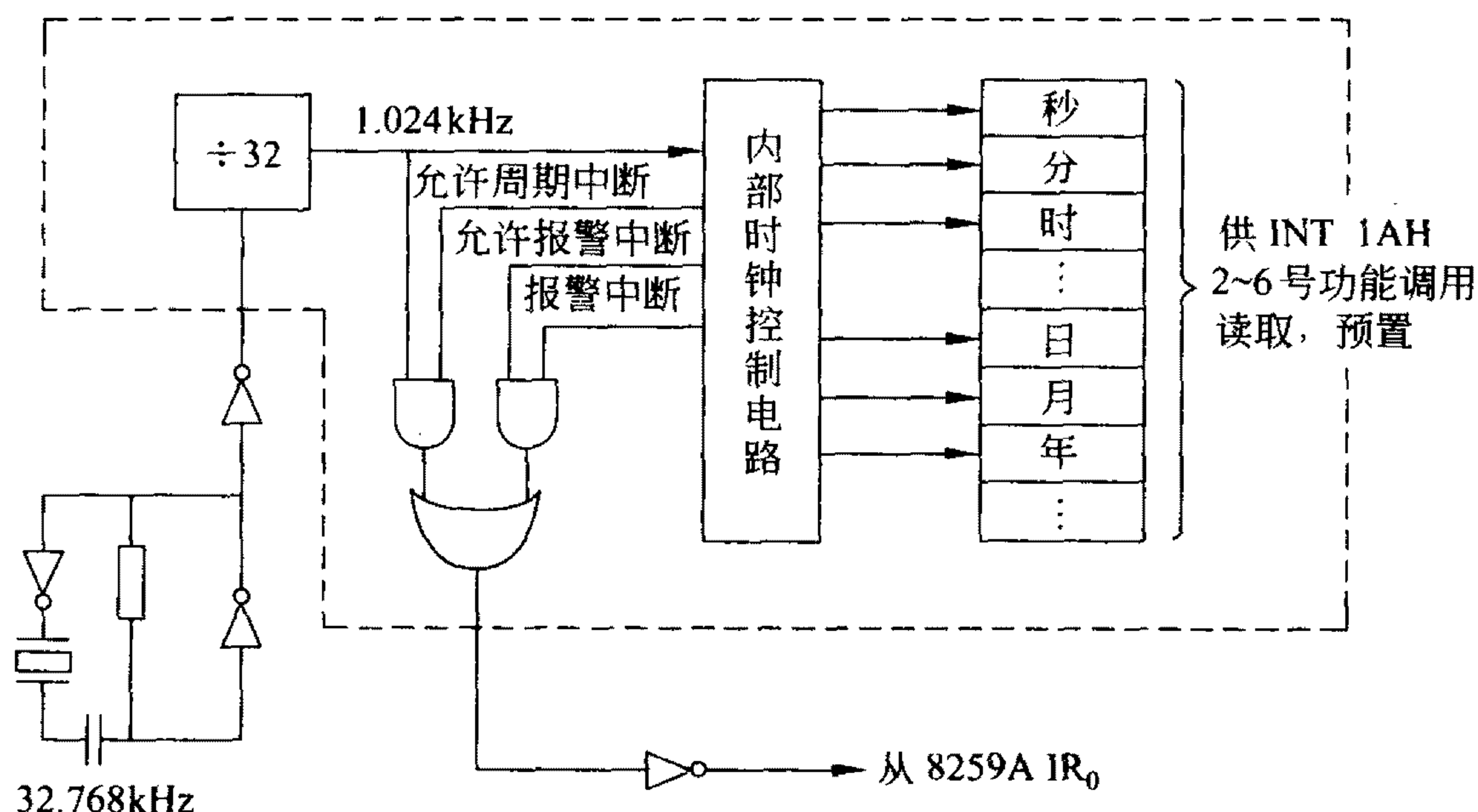


图 9-20 实时时钟电路示意图

1. 实时时钟电路工作原理

CMOS 内部偏移地址 0~9 单元存放着秒、报警秒、分、报警分、时、报警时、星期、日、月、年……信息,0AH~0DH 单元为内部的 A、B、C、D 4 个状态寄存器,0EH~3FH 存放系统配置信息,其中 32H 单元存放世纪信息。

基准频率为 32.768kHz,经过 CMOS RAM 内部 32 分频(这是初始化编程时设定的),产生了频率为 1.024kHz 的周期信号。在内部时钟控制电路作用下,每 1024 个周期对秒单元加 1,每 60 秒对分单元加 1……。

CMOS RAM 中的时间信息,可以通过 INT 1AH 的 2~6 号子功能读取或修改,用户程序一般不直接访问 CMOS。

系统加电后,CPU 执行 BIOS 中的一段程序,读取 CMOS 中的实时时钟信息,并转换成日时钟计数值,写入 40H:6CH~40H:6FH 的日时钟计数单元,做为日时钟计数器的计数初值,从而使日时钟和实时时钟同步。

该电路在系统断电后,由后备电池提供能源,继续工作,继续计时,故此称为实时时钟电路。

实时时钟电路为系统提供两种中断功能:

① 周期中断:在 CMOS 编程“允许周期中断”的前提下,每隔 $976.562\mu\text{s}$ (即

1/1024kHz)提出一次周期中断请求。

② 报警中断: 在 CMOS 编程“允许报警中断”的前提下, 当实时时钟达到预置的报警时间时, 产生报警中断。

需要注意的是: 系统启动后, BIOS 负责对 CMOS RAM 初始化, 初始化之后, CMOS RAM 禁止周期中断, 禁止报警中断。因此系统启动后, 没有实时时钟中断请求。

某个应用程序如果要开发周期中断、报警中断, 则由该应用程序负责重新对 CMOS RAM 编程, 使之允许周期中断, 或者允许报警中断。

周期中断和报警中断都从一个中断请求端引出, 接至从片 8259A 的 IR_0 。统称实时时钟中断。CPU 响应后自动转入实时时钟中断处理程序(即: 70H 型中断服务程序)。

2. 实时时钟中断处理流程

实时时钟中断处理程序主要做两件事:

① 读取 CMOS 状态寄存器, 判断是否是周期中断, 若是周期中断, 则“事件等待计数器”减去 $976\mu s$ ……, 不够减表明等待时间已到, 随即禁止周期中断(CMOS 从下一个 $976\mu s$ 开始, 就不再提出周期中断了), 并将“事件等待标志”置 0, “用户等待标志”置为 80H。

周期中断的这种功能为用户提供了定时操作的控制手段。用户程序可以查询“用户等待标志”, 若为 80H, 表明定时时间到, 转而执行预定的操作。

② 判断是否是报警中断, 若是, 则执行 INT 4AH。转入报警中断处理程序, 否则结束实时时钟中断处理。

读者一定会有许多问题:

① 既然在加电初始化时, CMOS 被禁止周期中断, 禁止报警中断, 那么什么时刻它才能被允许周期中断, 允许报警中断呢?

② 周期中断和报警中断有什么不同? 怎样开发?

③ 什么是用户等待标志, 什么是事件等待标志, 什么是事件等待计数器?

我们将在以后的内容中, 回答这些问题。图 9-21 为 BIOS 实时时钟中断处理程序的流程图。

9.9.2 周期中断

1. 周期中断流程

如果用户程序对 CMOS RAM 重新编程, 使 CMOS RAM 允许周期中断, 那么 CMOS RAM 将每隔 $976.562\mu s$ 有一次周期中断请求, CPU 响应周期中断之后自动转入 70H 型中断服务程序。由图 9-22 可知: 如果这次中断是由周期中断引发的, 则服务程序对“事件等待计数器”减 $976\mu s$, 再过 $976.562\mu s$ 又有一次周期中断, 服务程序对“事件等待计数器”再次减 $976\mu s$ ……。直到事件等待计数器不够减的时候, 服务程序使“用户等待标志置为 80H”, 并“禁止周期中断”, 此后, 如果不对 CMOS RAM 重新编程的话, CMOS RAM 就不再提出周期中断了。

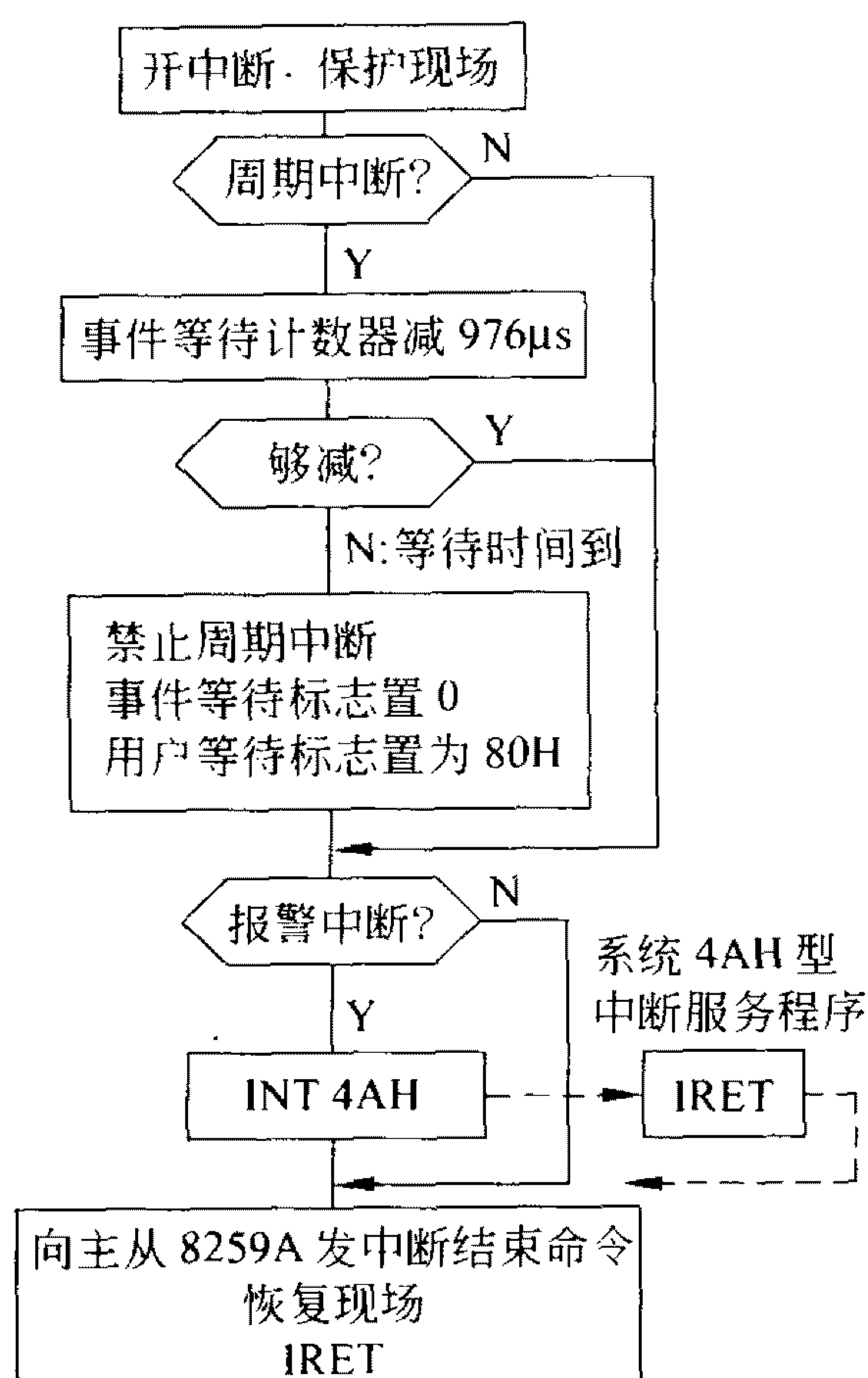


图 9-21 实时时钟中断处理流程
(BIOS 286 84/1/10 版)

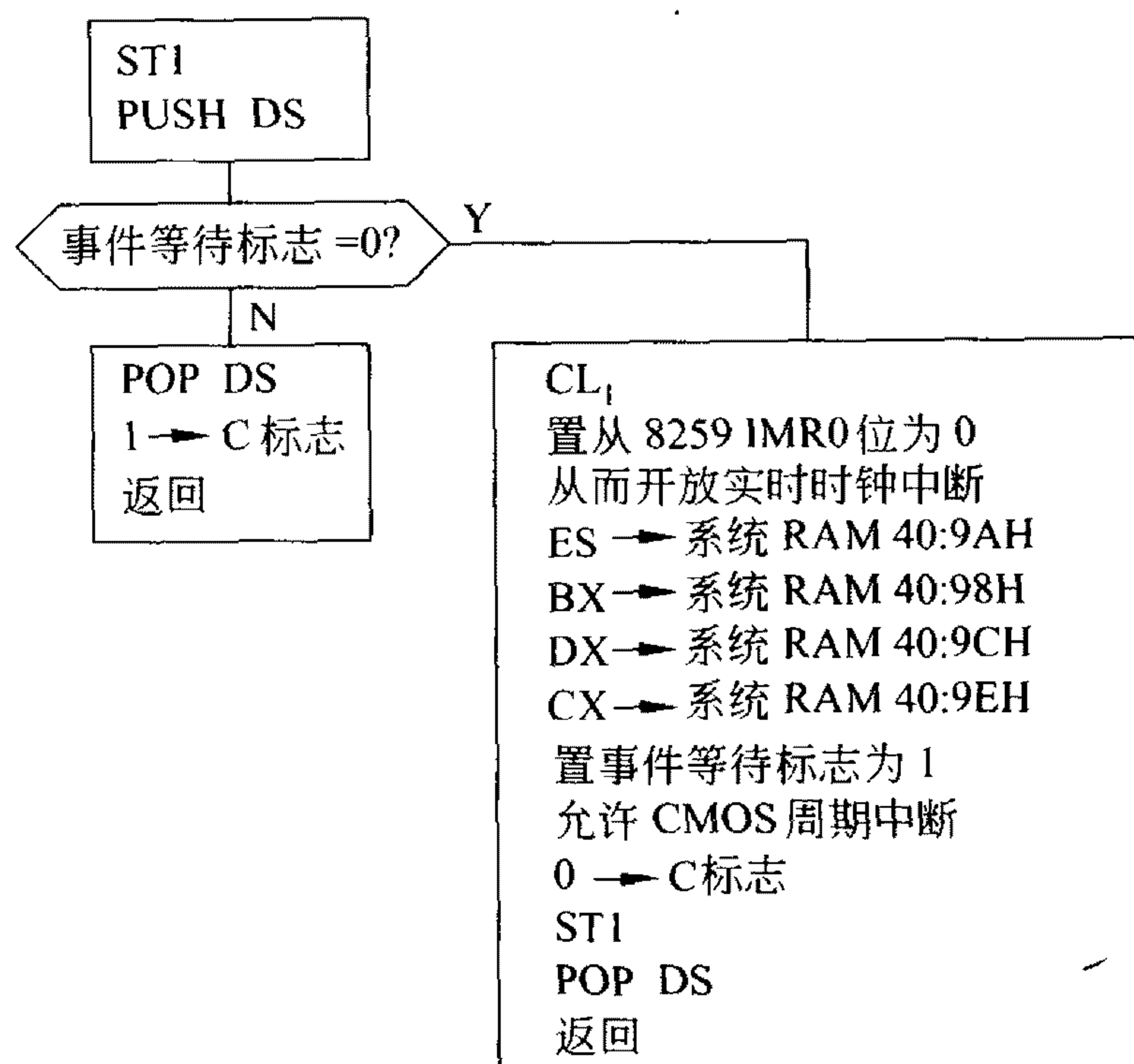


图 9-22 INT 15H 83H 子功能执行流程
(BIOS 286 84/1/10 版)

2. 事件等待计数器和用户等待标志

BIOS 规定：系统 RAM 40H: 9CH~40H: 9FH 这 4 个单元为事件等待计数器，用来存放用户预置的等待时间。用户等待标志是用户程序附加段的一个自定义单元，它是实时时钟中断处理程序（即 70H 型服务程序）向用户程序传递信息的单元。用户程序只要成功地预置了等待时间，并成功地使 CMOS RAM 允许周期中断，那么，当等待时间到的时候，实时时钟处理程序就会自动使用户等待标志置为 80H。

【例 9.8.1】和【例 9.8.2】利用日时钟中断，完成用户的定时操作——8 字循环左移。周期中断为用户程序完成定时操作提供了另一种方法——查询方式。用户可以利用周期中断开发程序，采用查询方式查询用户等待标志，若为 80H，表示等待时间到（即延时间到），从而转向预定的定时操作。

3. 周期中断的开发

欲开发周期中断，需要做两项准备工作：

- ① 对 CMOS 重新编程，使之允许周期中断。
- ② 预置等待时间。

这两项工作只需成功地调用 INT 15H 的 83H 子功能即可完成。

【INT 15H 83H 号子功能】预置等待时间
入口参数：

① 置 $AL=0$, 表明是预置等待时间;

② 置 CX, DX = 等待时间的微秒数, 其中 CX 为高 16 位二进制数, DX 为低 16 位二进制数。

③ 置 $ES: BX$ = 用户等待标志的逻辑地址。

出口参数:

C 标志置 1, 表示预置失败, C 标志置 0, 表示预置成功。

说明: 用户等待标志是一个字节型单元, 该单元要求设置在用户程序的附加段, 该单元偏移地址要求写入 BX 寄存器。

$INT\ 15H$ 的 $83H$ 号子功能执行流程如图 9-22 所示。

4. 相关的 BIOS 工作单元

BIOS 规定:

① 系统 $RAM\ 40H: A0H$ 单元为“事件等待标志”, 它和“用户等待标志”无关, 是为了 BIOS 程序设计的方便而设置的工作单元, 该单元由 $INT\ 15H$ 的 $83H$ 号子功能置 1, 实时时钟中断处理程序置 0, 只有当“事件等待标志”为 0 时, 用户才可以成功地预置等待时间。但是, 用户程序不必查询“事件等待标志”。

② 系统 $RAM\ 40H: 98H \sim 40H: 9BH$ 存放“用户等待标志”的逻辑地址。

③ 系统 $RAM\ 40H: 9CH \sim 40H: 9FH$ 存放用户预置的等待时间, 这 4 个字节就是前文所说的“事件等待计数器”。

若成功的执行了 $INT\ 15H$ 的 $83H$ 号子功能, 意味着: BIOS 已经对 CMOS 重新编程, 允许周期中断。打这以后, CMOS 才能每隔 $976\mu s$ 提出一次周期中断请求。CPU 响应后转入实时时钟中断处理, 对事件等待计数器减 $976\mu s \dots$, 不够减时, 置“用户等待标志”为 $80H$, 通报等待时间到, 用户程序查询“用户等待标志”为 $80H$ 时, 转而执行预定的操作。执行流程如图 9-22 所示。

【例 9.9.1】 试验周期中断。

编写一个程序, 利用周期中断, 在程序执行后每隔 2 秒钟, 显示一串字符“TIME TO!”按任意键时停止。

【程序清单】

```

;FILENAME: 991. ASM
DATA SEGMENT
FLAG DB 00H ;用户等待标志
TTT DD 2000000 ;等待时间(微秒)
MSG DB 'TIME TO !', 0DH, 0AH, '$'
DATA ENDS
CODE SEGMENT
ASSUME CS: CODE, DS: DATA, ES: DATA
BEG: MOV AX, DATA
MOV DS, AX
MOV ES, AX

```

```

AGA:      MOV      AH,83H          ;预置等待时间
          MOV      AL,0
          MOV      BX,OFFSET FLAG   ;ES: BX=用户等待标志的逻辑地址
          MOV      CX,WORD PTR TTT+2 ;予置等待时间
          MOV      DX,WORD PTR TTT   ;为 2 秒钟
          INT      15H
          JC       AGA
          STI
SCAN:     MOV      AH,1
          INT      16H
          JNZ      EXIT             ;有键入转移
          CMP      FLAG,80H         ;查询用户等待标志
          JNZ      SCAN             ;用户等待标志没有置位转移
DISP:     MOV      AH,9
          MOV      DX,OFFSET MSG
          INT      21H
          MOV      FLAG,0           ;用户等待标志复位
          JMP      AGA              ;第 29 行指令
EXIT:     MOV      AH,4CH
          INT      21H
CODE      ENDS
          END      BEG

```

【程序分析】

① 程序执行后,每隔 2 秒钟,屏幕显示一行“TIME TO!”。

② 如果第 29 行指令改成 JMP SCAN,屏幕只能显示一行 TIME TO! 这说明用户等待标志的置位是一次性的,分析实时时钟中断处理流程可以证实这一点。因为等待时间到的时候,BIOS 重新对 CMOS 编程使其禁止周期中断,如果欲使“用户等待标志”周期性地置位,那么必须周期性地预置等待时间。本程序第 29 行现为 JMP AGA 就达到了这一目的。

【程序说明】

INT 15H 的 83H 子程序管理一个“事件等待计数器”,该子功能一次只能供一个进程使用,如果有别的进程正在使用该功能,又有另一个程序企图调用该项功能,则该项功能调用后,C 标志为 1,表示调用失败。

成功地调用 INT 15H 的 83H 子程序,是开发周期中断的关键。在 MS-DOS 以及 Windows 98(含 98)以下的 DOS 方式,用户程序均能成功地调用该项功能。但遗憾的是在 Windows 2000 以上操作系统下的“命令提示符”方式,用户程序无法成功地调用 INT 15H 的 83H 子程序。实验证明,在这种环境下,用户程序无法成功地预置周期中断的等待时间,无法执行上述程序。

9.9.3 报警中断

1. 报警中断的开发

在 CMOS 编程允许报警中断的前提下,当实时时钟达到用户预置的报警时间的时

候, CMOS 发出报警中断。CPU 响应后最终转向报警中断处理程序(即 4AH 型中断服务程序)。

欲开发报警中断需做两项准备工作:

- ① 对 CMOS 重新编程使之允许报警中断。
- ② 预置报警时间。

这两项工作只需先行调用 INT 1AH 的 07H 号子功能,然后调用 INT 1AH 的 06H 号子功能,即可完成。

【INT 1AH 的 07H 号子功能】复位报警,为预置报警时间做准备。

入口参数:无。

【INT 1AH 的 06H 号子功能】预置报警时间。

入口参数:

- ① 置 CH=报警时刻的小时数 (0~23 的 BCD 码数)。
- ② 置 CL=报警时刻的分钟数 (0~59 的 BCD 码数)。
- ③ 置 DH=报警时刻的秒数 (0~59 的 BCD 码数)。

出口参数:

C 标志置 1,表示预置失败;C 标志置 0,表示预置成功。

INT 1AH 的 06H 号子功能完成如下操作:

① 读取 CMOS 状态寄存器,判断是否允许报警中断,若是表明曾经预置过报警时间,则本次预置无效,置 C 标志为 1 返回。假若状态寄存器表明已禁止报警中断,再执行以下操作。

② 把 CH、CL、DH 中的报警时、分、秒写入 CMOS RAM 偏移地址 5、3、1 的单元中。

③ 置从片 8259A 的 IMR0 位为 0,开放实时时钟中断。

④ 对 CMOS 重新编程,允许报警中断。

用户成功地预置了报警时间后,当实时时钟达到报警时间时,CMOS 提出报警中断。CPU 响应后,转入实时时钟中断处理程序,再由后者执行 INT 4AH,转入报警中断服务程序。

注意: BIOS 为报警中断设计的服务程序,只有一条 IRET 指令,用户应自行设计报警中断服务程序,取代原先的 4AH 型服务程序。

2. 周期中断与报警中断的区别

从以上论述可以看出,报警中断也为用户提供了一个时间等待的功能。在这一点上和周期中断十分相似,但是报警中断和周期中断有两点不同:

- ① 时间概念不同。

用户在开发周期中断之前,要预置等待时间,这个时间是相对时间,是相对于当前(程序执行时)的实时时钟而言的。

用户在开发报警中断之前,要预置报警时间,这个时间是绝对时间。

- ② “时间到”的通报方式不同。

对于周期中断而言,当等待时间到时,BIOS 置用户等待标志为 80H。用户程序只能

用查询方式,查询等待标志,从而转向预定的操作。

对于报警中断而言,当预定的报警时间到时,CPU 最终转向 4AH 型中断服务程序。所以用户程序可以用中断方式响应。

【例 9.9.2】 试验报警中断。

设置报警时间,当实时时钟到报警时间时,喇叭发出 3 声短促的音响。

【程序清单】

```

;FILENAME: 992. ASM
CODE      SEGMENT
          ASSUME  CS: CODE
BEG:      CLI
          CALL    WRITE4A
          CALL    C_ALARM      ;清除报警时间
          CALL    S_ALARM      ;设置报警时间
          CALL    SETTIME      ;设置系统时间
          STI
SCAN:     MOV    AH,1
          INT     16H
          JZ      SCAN         ;等待中断
          CALL    C_ALARM      ;清除报警时间
          MOV     AH,4CH
          INT     21H

;-----
SERVICE  PROC
          MOV     CX,3
OPEN:     IN      AL,61H
          OR      AL,00000011B
          OUT     61H,AL       ;接通扬声器
          CALL    DELAY        ;延时
CLOSE:    IN      AL,61H
          AND     AL,11111100B
          OUT     61H,AL       ;关闭扬声器
          CALL    DELAY        ;延时
          LOOP    OPEN
          IRET
SERVICE  ENDP

;-----
DELAY     PROC
          PUSH    CX
          MOV     AH,2DH
          MOV     CX,0
          MOV     DX,0
          INT     21H
READ:     MOV     AH,2CH

```

```

        INT      21H
        CMP      DL,10
        JC       READ          ;小于 10 个百分秒转移
        POP      CX
        RET
DELAY    ENDP
;-----
WRITE4A  PROC          ;设置 4AH 型中断向量
        MOV      AX,CODE
        MOV      DS,AX
        MOV      DX,OFFSET SERVICE
        MOV      AX,254AH
        INT      21H
        RET
WRITE4A  ENDP
;-----
SETTIME  PROC          ;设置系统时间
AGA:     MOV      AH,03H
        MOV      CX,1030H
        MOV      DX,0
        INT      1AH
        JC       AGA
        RET
SETTIME  ENDP
;-----
C_ALARM  PROC          ;清除报警时间
        MOV      AH,07H
        INT      1AH
        RET
C_ALARM  ENDP
;-----
S_ALARM  PROC          ;设置报警时间
AGAIN:   MOV      AH,06H
        MOV      CX,1030H
        MOV      DH,05H
        INT      1AH
        JC       AGAIN
        RET
S_ALARM  ENDP
CODE     ENDS
        END      BEG
```

【程序说明】

① SETTIME 子程序调用 1NT 1AH 的 3 号子功能设置系统实时时钟为 10 时 30 分

00 秒,这是为了试验方便而设计的。

② CLRALARM 子程序复位报警。

③ SETALARM 子程序预置报警时间为 10 时 30 分 05 秒,这样程序执行后 5 秒钟即可听到 3 声短促的音响。

④ WRITE4A 子程序,写入用户的服务程序中断向量,取代系统的 4AH 型中断向量。

⑤ SERVICE 服务子程序,使喇叭发出 3 声短促的音响。

9.10 键盘中断

9.10.1 键盘中断全过程

键盘是微型计算机系统的输入设备,早期的 PC、PC/XT 机多使用 83 键、84 键,所谓“标准键盘”,286 以上微型计算机多使用 101 键、102 键等“扩展键盘”,它们都是非编码键盘。

键盘接口包括两部分,一部分组装在键盘盒内部,称为键盘电路。另一部分安装在主机板上,称为键盘接口电路。键盘与主机之间用 5 芯电缆连接,这 5 根芯线是:电源(+5V)、地线、时钟、信号线和备用线。

键盘电路是以单片机为核心的键盘扫描电路。系统加电后固化在单片机中的键盘扫描程序,周期性地扫视每一个按键,一旦有按键按下,键盘扫描程序立即识别闭合键的行列位置,然后通过信号线串行地发出闭合键的接通扫描码。闭合键断开后,键盘电路又发出该键的断开扫描码。

与扩展键盘配套的键盘接口电路,用单片机做为接口控制器,在单片机软件的支持下,接收来自键盘电路的按键扫描码;对串行数据进行奇偶校验;完成串→并转换;将表示行列位置的扫描码转换成系统扫描码。转换完毕由键盘接口电路向主 8259A IR₁ 端子提请中断。

CPU 响应键盘中断后,转入键盘中断处理程序,主要完成以下工作:开中断、保护现场、从键盘接口电路(端口地址 60H)读取按键扫描码,对扫描码进行分析、处理,最终生成相应的键代码存入键盘缓冲区,最后向主 8259A 发出中断结束命令,恢复现场,中断返回。

键盘缓冲区,位于系统 RAM 40H:1EH~40H:3DH 共 32 个单元,实际使用 30 个单元存放 15 个键的键代码,缓冲区是一个环形的队列结构,数据存取遵循“先进先出”的规律,键盘缓冲区中的键代码由 9 型中断服务程序负责写入,用户程序调用 INT 16H 可以读取键盘缓冲区中的信息。因此键盘缓冲区是键盘硬中断(即 9 型中断服务程序)和 INT 16H 软中断之间传递信息的“缓冲区”。图 9-23 画出了键盘中断全过程示意图。

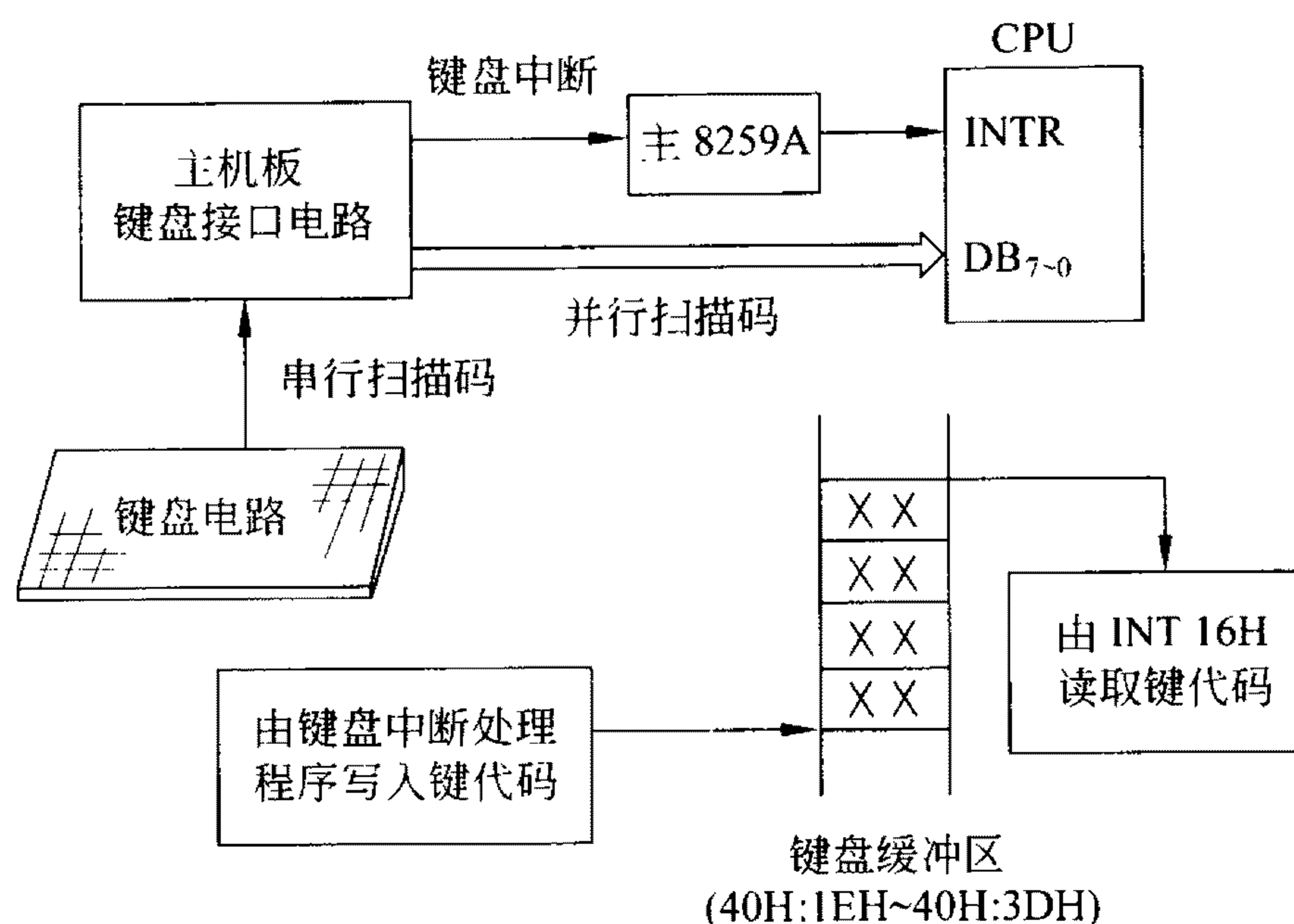


图 9-23 键盘中断全过程示意图

9.10.2 键代码生成

1. 特殊键状态标志

键盘中断处理程序,从键盘接口电路获取按键扫描码,对其进行分析,区分是字符键还是特殊键。

特殊键是: Ins, Caps Lock, Num Lock, Scroll Lock, Alt, Ctrl, 左、右 Shift, 前四个是开关键。

BIOS 规定: 系统 RAM 40H: 17H 单元为特殊键的键标志单元。用来记录特殊键的状态。键标志单元位结构如下:

D ₇ : Ins 键	奇数次按下置 1, 偶数次按下置 0。
D ₆ : Caps Lock 键	奇数次按下置 1, 偶数次按下置 0。
D ₅ : Num Lock 键	奇数次按下置 1, 偶数次按下置 0。
D ₄ : Scroll Lock 键	奇数次按下置 1, 偶数次按下置 0。
D ₃ : Alt 键	按下置 1, 抬起置 0。
D ₂ : Ctrl 键	按下置 1, 抬起置 0。
D ₁ : 左 Shift 键	按下置 1, 抬起置 0。
D ₀ : 右 Shift 键	按下置 1, 抬起置 0。

用户程序对 40H: 17H 单元直接访问, 或者通过 INT 16H 的 02H 号功能可以读取键标志单元的信息(101 键扩展键盘使用 INT 16H 的 12H 号功能调用)。

2. 字符键键代码

键盘中断处理程序判断出按键扫描码是字符键之后, 再根据 Alt、Ctrl、Shift 的状态标志, 生成两字节的键代码存入键盘缓冲区, 用户程序调用 INT 16H 的 10H 号子程序可以从键盘缓冲区取出键代码→AX 寄存器。

下面列举 3 种键代码格式：

- ① 对于单一的字符键，键代码高位字节为系统扫描码，低位字节为代表键含意的标准 ASCII 码。
- ② 对于功能键 F1~F12，以及特殊键和功能键的组合，生成的键代码高位字节为扩展码，低位字节为 0。表 9-2 仅列出了功能键、光标键和部分组合键的键代码。

表 9-2 功能键、光标键和部分组合键键代码(H)

按键	键代码	按键	键代码	按键	键代码
F1~F10	3B00~4400	F11	8500	F12	8600
Shift+F1~F10	5400~5D00	Shift+F11	8700	Shift+F12	8800
Ctrl+F1~F10	5E00~6700	Ctrl+F11	8900	Ctrl+F12	8A00
Alt+F1~F10	6800~7100	Alt+F11	8B00	Alt+F12	8C00
↑	48E0	↓	50E0	←	4BE0
Ctrl+↑	8DE0	Ctrl+↓	91E0	Ctrl+←	73E0
→	4DE0	Ctrl+→	74E0		

- ③ 小键盘操作生成的键代码高位字节为 0，低位字节为 1~255。

小键盘操作输入数据的步骤是：先按下 Alt 键(不要松开)再按下小键盘数字键，当 Alt 键松开后，即可完成 1~255 数字的输入。同时也在屏幕上生成相应的 CRT 显示符。图 9-24 是小键盘操作画出的单线框和双线框。

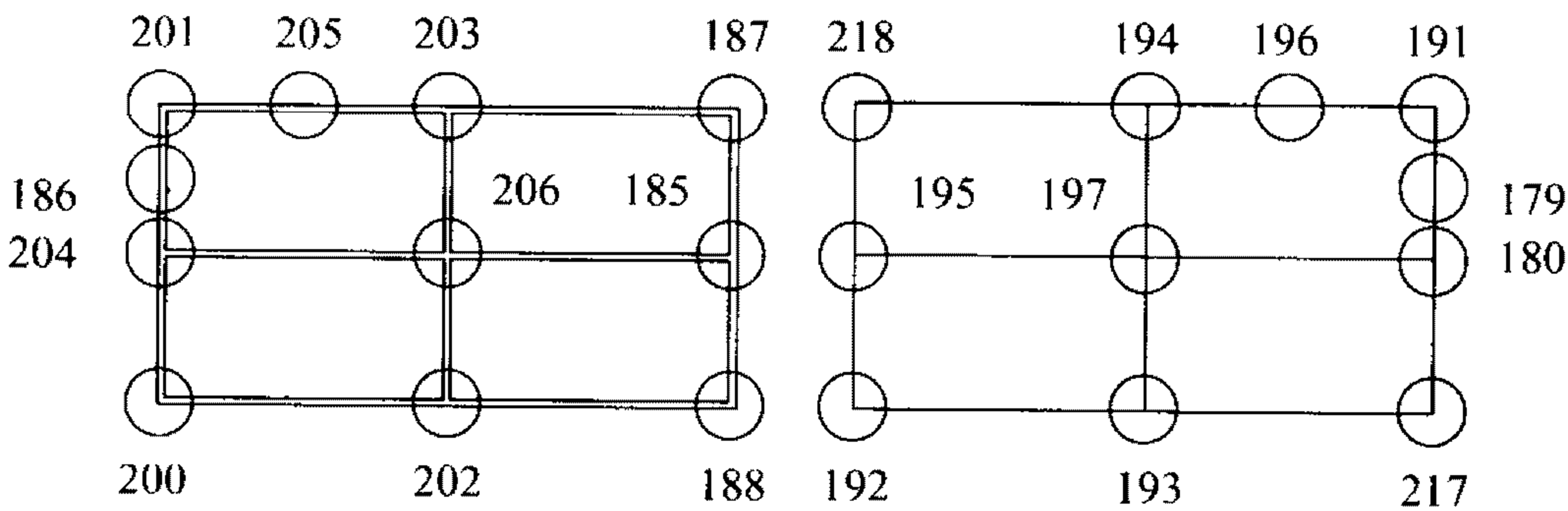


图 9-24 小键盘操作画出的单线框和双线框

3. 键代码测试

编程中你可能要屏显一些特殊的符号(例如：黑桃、方块、梅花……)，也许要用到一些特殊的组合键，运行下面两个程序可以得到特殊符号的编码或者组合键的键代码。

【例 9.10.1】 特殊符号显示程序。

下列程序执行后，每按一次 Enter 键，屏幕上就显示一个符号和该符号的编码，按 Esc 键之后程序结束。

【程序清单】

```
;FILENAME: CRTDISP. ASM
```



```

DATA      SEGMENT
MSG      DB      '-----',?,?,',H',0DH,0AH,'$'
KEYCODE  DB      1
DATA      ENDS
CODE      SEGMENT
          ASSUME  CS: CODE,DS: DATA
BEG:      MOV     AX,DATA
          MOV     DS,AX
LAST:     MOV     AH,08H
          INT     21H                ;等待键入
          CMP     AL,1BH
          JE      EXIT              ;是“Esc”转移
          CMP     AL,0DH
          JNE     LAST              ;不是回车转移
          MOV     AH,0EH
          MOV     AL,KEYCODE
          INT     10H                ;显示一个符号
          CALL    N2_16              ;调用二→十六进制显示子程序
          INC     KEYCODE
          JMP     LAST
EXIT:     MOV     AH,4CH
          INT     21H
;-----
N2_16     PROC                                ;二→十六进制
          MOV     BX,OFFSET MSG+5
          MOV     AH,KEYCODE
          MOV     CX,2
AGA:      ROL     AX,4
          AND     AL,0FH
          CMP     AL,10
          JC      NEXT2
          ADD     AL,7
NEXT2:    ADD     AL,30H
          MOV     [BX],AL
          INC     BX
          LOOP    AGA
          MOV     AH,9
          MOV     DX,OFFSET MSG
          INT     21H
          RET
N2_16     ENDP
CODE      ENDS
          END     BEG

```

【例 9.10.2】 键代码测试程序。

下面程序运行后,键入任意键,或任意组合键,即可显示按键的键代码,该程序按 Esc 键之后,先显示键代码,然后结束程序。

【程序清单】

```

;FILENAME: KEYCODE. ASM
. 486
CODE    SEGMENT  USE16
        ASSUME   CS:CODE
BEG:    MOV      AH,11H
        INT      16H
        JZ       BEG           ;无键入转移
        MOV      AH,10H
        INT      16H           ;取键代码→AX
        MOV      BL,AL
NEXT1:  SAL      EAX,16
        CALL     N2_16         ;显示高位字节
        MOV      AH,0EH
        MOV      AL,'/'
        INT      10H
        CALL     N2_16         ;显示低位字节
        MOV      CX,3
LAST:   MOV      AH,0EH
        MOV      AL,' '
        INT      10H
        LOOP     LAST
        CMP      BL,1BH        ;是'Esc'吗?
        JNE      BEG
EXIT:   MOV      AH,4CH
        INT      21H
;-----
N2_16   PROC           ;二→十六进制
        MOV      CX,2
AGA:    ROL      EAX,4
        AND      AL,0FH
        CMP      AL,10
        JC       NEXT2
        ADD      AL,7
NEXT2:  ADD      AL,30H
        MOV      AH,0EH
        INT      10H           ;显示
        LOOP     AGA
        RET
N2_16   ENDP
CODE    ENDS
        END      BEG

```




9.11 驻留程序

用户程序通常采用 4CH 功能调用返回 DOS, 该项功能调用, 终止一个程序的运行, 并把控制权转交 DOS, 同时也把用户程序占用的内存空间交还 DOS, 由 DOS 另行分配。从程序员的角度来看, 一个程序执行完了, 返回 DOS 之后, 程序在内存中就不复存在, 若想再次运行, 就需要再次把它调入内存。

驻留程序(Terminate and Stay Resident Program, TSR 程序)是另一回事。TSR 程序是特殊的中断服务程序, TSR 程序常驻内存, 它占用的内存空间受到 DOS 的保护, 不被后续装入的程序覆盖。TSR 程序平时“潜伏”在后台, 处于休闲状态, 一旦激活, 它将中断前台的应用程序。系统中有许多输入输出驱动程序都是常驻内存的, 用户也可以设计 TSR 程序。

驻留程序增强了微型计算机系统的功能, 从宏观上讲, CPU 可以“同时执行”许多程序, 其中之一为前台程序, 其他是各种各样的驻留程序。

设计 TSR 程序要考虑的问题很多, 下面仅就一些主要问题做浅显的分析。

本节首先介绍驻留程序的设计方法, 然后论述解除驻留的机理, 在此基础上再介绍如何设计有解驻功能的程序。

9.11.1 驻留程序的设计方法

1. 怎样实现程序驻留

DOS 为实现程序驻留提供了两种方法:

① DOS 中断 INT 27H

入口参数: CS=被驻留程序的 PSP 段基址, 也即程序被装入时的 DS 或 ES 值。

DX=被驻留程序的字节数, 包括被驻留程序的 PSP 段。

出口参数: 无。

使用 INT 27H, 实现程序驻留比较简便, 它特别适合按照 COM 格式编写的源程序, INT 27H 可驻留的目标代码体积小于 64KB-100H。

② INT 21H 的 31H 号功能

入口参数: AH=31H。

DX=被驻留程序的体积(包括被驻留程序的 PSP 段), 单位是“节”, (一节等于 16 个字节)。

出口参数: 无。

2. 驻留程序的框架结构

驻留程序可以按 EXE 格式也可以按 COM 格式编程, 下面给出的是按 COM 格式编程的驻留程序的框架结构:

```

.486
CODE SEGMENT USE16
    ASSUME CS: CODE,DS: CODE
    ORG 100H
BEG: JMP INIT
    驻留程序数据区
TSR1 PROC
    待驻留的程序
TSR1 ENDP
TSR2 PROC
    待驻留的程序
TSR2 ENDP
INIT: 准备工作                                ;INIT 以下为初始化程序
      MOV DX,OFFSET INIT                    ;计算驻留程序的体积→DX
      INT 27H                               ;驻留返回
CODE ENDS
      END BEG

```

该程序调入内存后,从 BEG 单元跳转到 INIT 单元执行驻留程序的初始化部分,随后执行 INT 27H 驻留返回。标号 INIT 以上的部分(包括该程序的 PSP 段)被驻留在内存之中,INIT 以下的部分被后续装入的其他程序覆盖。

驻留程序由若干个中断服务子程序组成,服务程序结束有两种返回方式:

- ① 执行“IRET”指令返回到被中断的前台程序的断点。
- ② 执行“JMP 原中断向量”转移到被驻留程序替代的系统中断服务程序。

3. 驻留程序的激活方式

上面讲到:驻留程序是由若干个中断服务程序组成的,因为是中断服务程序(不是一般的子程序),因此,只有在发生中断的时候才去执行它,怎样引导 CPU 执行它们呢?通常采用的激活方式有以下几种:

(1) 执行软中断指令 INT n 激活

被激活的驻留程序,必须是 n 型中断服务程序,它必须挂接在 n 型中断链上。“挂接”这一术语的含义是:把驻留程序中的某个中断服务程序入口地址写入到中断向量表的 4×n~4n+3 单元中,挂接操作在初始化程序的准备工作中完成。

(2) 时钟中断激活

用这种方式激活的驻留程序,应当挂接在 8 型中断或者 1CH 型中断链上,驻留程序完成定时操作。

(3) 热键激活

热键激活是驻留程序最常用的激活方式。“热键”是一般应用程序很少使用的冷门键,例如键盘上的功能键、开关键、特殊组合键等。

4. 怎样获取热键信息

获取热键信息的方法是:设计“热键识别程序”(又称键盘中断扩充程序),将它挂接

在 9 型中断链上。每当有按键操作时, CPU 即转而执行热键识别程序。该程序用以下两种方法之一识别热键。

① 进入热键识别程序以后, 直接从键盘接口电路端口地址 60H 中读取信息, 如果是热键则给出激活标志, 否则不设置激活标志。这种热键识别程序, 其出口指令为“JMP 原 9 型中断向量”。

在键盘中断一节中介绍过, 键盘电路把闭合键扫描码串行地传送给键盘接口, 后者把串行扫描码转换成并行扫描码, 然后向主 8259A IR_1 端子提请中断, CPU 响应键盘中断后, 转入键盘中断服务程序, 该程序从键盘接口电路端口地址为 60H 的寄存器中取出并行扫描码进行分析、处理, 最终生成相应的键代码存入键盘缓冲区。如果热键识别程序是从 60H 端口寄存器中获取信息的话, 则获取的信息是按键的扫描码而不是按键的键代码。显然这一操作执行完毕应当转入系统的键盘中断服务程序再由键盘中断服务程序生成按键的键代码。

② 进入热键识别程序以后, 执行“CLI”, “PUSHF”和“CALL 原 9 型中断向量”3 条指令, 先转入老的 9 型中断服务程序, 由后者生成按键的键代码存入键盘缓冲区, CPU 从老的 9 型中断服务程序返回之后, 再由热键识别程序从键盘缓冲区取出按键的键代码判断, 如果是热键则给出激活标志, 否则不给出激活标志。这种热键识别程序其出口指令为“IRET”。

5. 热键信息的处理

获取热键信息之后, 首要的任务是激活驻留程序中相关的中断服务程序, 对于热键信息本身如何处理呢? 众说纷纭, 不外乎两种选择:

① 截留。在完成激活任务之后不让热键的键代码流入键盘缓冲区, 或者从键盘缓冲区中清除热键的键代码, 最终目的是不让前台程序使用。

② 不截留。完成激活任务之后, 热键的键代码仍然保留在键盘缓冲区中, 供前台程序使用。

6. 避免“DOS 重入”

驻留程序的激活是随机的, 在驻留程序激活后, 它所中断的前台程序是无法预料的。当前台程序正在执行“INT 21H”的某项功能时, 驻留程序被激活, 如果驻留程序本身也使用“INT 21H”, 可能造成 DOS 重入。

由于 DOS 是单任务操作系统, “INT 21H”的大部分功能调用是不允许重入的。如果强行 DOS 重入, 很可能使系统瘫痪。

① 避免 DOS 重入最简单的对策是: 在驻留程序中避免使用 DOS 功能调用。如果驻留程序涉及键盘输入、屏幕显示、字符打印等项操作, 应当避免使用“INT 21H”的 1~0CH 功能调用, 改换成相应的 BIOS 功能调用。

举例来讲, 如果驻留程序需要向屏幕输出一串字符, 用“INT 21H”的 9 号功能, 很可能造成 DOS 重入而死机。改用“INT 10H”, 用它的 13H 号功能, 或者直接对“视屏映像区”写入, 就不会产生 DOS 重入的灾难了。

② 避免 DOS 重入最稳妥的办法是,在中断程序中查询“DOS 忙闲标志”,在确保 DOS 空闲的条件下再调用 DOS 功能。

DOS 忙闲标志占用一个字节(称为 INDOS 单元),该字节为 0 表示 DOS 空闲,非 0 表示忙。DOS 有一个 34H 号功能调用,它提供了 DOS 忙闲标志的逻辑地址。

该项功能调用格式如下:

入口参数: AH=34H。

出口参数: ES: BX=DOS 忙闲标志所在单元的段基址: 偏移地址。

在驻留程序的初始化部分(这一部分是不被驻留的)使用“INT 21H”的 34H 号功能,读取 DOS 忙闲标志的逻辑地址,并存入驻留程序的数据单元,在驻留程序激活后,进行 DOS 服务之前,再根据获取的地址信息,查询 DOS 忙闲标志,在确保 DOS 空闲的条件下,再调用 DOS 功能。

9.11.2 驻留程序设计举例

驻留程序的设计比较麻烦,很难做到一气呵成。本小节以驻留一个音响程序为例由浅入深地介绍程序设计的全过程,【例 9.11.1】、【例 9.11.2】仅仅是前导例题,【例 9.11.3】完成最终设计。

【例 9.11.1】 非驻留式的音响程序。

该程序按 COM 格式编程,程序执行后发出 3 声短促的音响,然后返回 DOS,程序不驻留内存。

【程序清单】

```

;FILENAME: 9111. ASM
DELAY      MACRO
            LOCAL    LAST1,LAST2
            MOV      AX,200
LAST1:     MOV      BX,65535
LAST2:     DEC      BX
            JNZ      LAST2
            DEC      AX
            JNZ      LAST1
            ENDM
;-----
.486
CODE       SEGMENT USE16
            ASSUME   CS: CODE
            ORG      100H
BEG:       MOV      CX,3
OPEN:      IN       AL,61H
            OR       AL,03H
            OUT      61H,AL          ;接通扬声器
            DELAY                    ;延时

```



```

CLOSE:      IN      AL,61H
            AND      AL,11111100B
            OUT      61H,AL          ;关闭扬声器
            DELAY                    ;延时
            LOOP     OPEN
            INT      20H
CODE        ENDS
            END      BEG

```

【例 9.11.2】 驻留式音响程序。

将【例 9.11.1】的程序改造成驻留程序,程序驻留后采用热键(Ctrl+F11)激活,发出 3 声短促的音响,若再次激活则再次发出 3 声音响。程序没有解除驻留的功能。

【程序清单】

```

;FILENAME: 9112. ASM
.486
CODE    SEGMENT USE16
        ASSUME  CS: CODE
        ORG     100H
BEG:     JMP     INIT
OLD09    DD     ?          ;存放系统 9 型中断向量
OLD1C    DD     ?          ;存放系统 1CH 型中断向量
HOTKEY   DB     0          ;存放激活标志
COUNT   DB     6
NEW09    PROC
        CLI
        PUSHF
        CALL    OLD09      ;转系统 9 型中断服务程序
        PUSHA                    ;保护现场
        MOV     AH,11H
        INT     16H
        JZ      EXIT_1      ;无键入转
        CMP     AX, 8900H    ;Ctrl_F11?
        JNE     EXIT_1      ;不是,转
NEXT1:   MOV     HOTKEY,-1    ;设置激活标志
        MOV     AH,10H
        INT     16H          ;清除热键信息
EXIT_1:  POPA                    ;恢复现场
        IRET                    ;中断返回
NEW09    ENDP
;-----
NEW1C    PROC
        CMP     HOTKEY,-1    ;有激活标志吗?
        JNE     EXIT_2      ;没有转
ALARM:   IN      AL,61H

```

```

        XOR        AL,00000011B
        OUT        61H,AL          ;打开或者关闭扬声器
        DEC        COUNT          ;计数
        JNZ        EXIT_2         ;不满 6 次转
        MOV        COUNT,6
        MOV        HOTKEY,0       ;清除激活标志
EXIT_2:  IRET              ;中断返回
NEW1C   ENDP
;-----
INIT:   MOV        AX,3509H        ;置换 9 型中断向量
        INT        21H
        MOV        WORD PTR OLD09,BX
        MOV        WORD PTR OLD09+2,ES
        MOV        AX,2509H
        MOV        DX,OFFSET NEW09
        INT        21H
        MOV        AX,351CH        ;置换 1CH 型中断向量
        INT        21H
        MOV        WORD PTR OLD1C,BX
        MOV        WORD PTR OLD1C+2,ES
        MOV        AX,251CH
        MOV        DX,OFFSET NEW1C
        INT        21H
        MOV        DX,OFFSET INIT
        INT        27H              ;驻留退出
CODE    ENDS
        END        BEG

```

【程序分析】

驻留程序中设计了一个热键识别程序,它挂接在 9 型中断链上,取名为“NEW09”。在驻留程序的初始化部分,首先将热键识别程序的入口地址填写到 $4 \times 9 \sim 4 \times 9 + 3$ 的中断向量表单元,取代系统原来的 9 型中断向量,程序驻留内存之后,一旦有键盘输入,CPU 响应后首先转入热键识别程序,在那里,顺序执行“CLI”,“PUSHF”和“CALL 原 9 型中断向量”3 条指令,调用老的键盘中断服务程序,由后者生成按键的键代码存入键盘缓冲区,CPU 从老的键盘中断服务程序返回后,程序再调用“INT 16H”的 11H 号子程序测试键代码,如果是 Ctrl+F11 就令热键标志单元(HOTKEY 单元)为 -1 作为激活标志。

驻留程序中还设计了一个过程名为“NEW1C”的中断服务程序,它挂接在 1CH 型中断链上,在驻留程序的初始化部分将它的入口地址填写到 $4 \times 1CH \sim 4 \times 1CH \times 3$ 单元,取代系统原来的 1CH 型中断向量。程序驻留内存之后,每隔 55ms 通过 8 型中断转而执行新的 1CH 型中断服务程序,后者查询 HOTKEY 单元,若有激活标志则打开或者关闭扬声器,该子程序被调用 6 次,可以使扬声器发出 3 声短促的音响,然后清除热键信息返

回系统原来的 8 型中断服务程序。

9.11.3 驻留程序的解驻

对于驻留程序,要想解除驻留怎么办?解驻要做两件事:其一,把驻留程序从它挂接的中断链上“卸”下来,就是讲要恢复被驻留程序置换的全部中断向量,这一操作称为“卸载”;其二,释放驻留程序占用的全部内存空间(包括它的 PSP 段和环境变量区)这一操作称为“解驻”。

卸载比较容易。要想释放被驻留程序占用的内存空间,则必须首先了解它占用了哪些空间,为此要了解 DOS 的内存管理。DOS 的内存管理是一项很复杂的技术,本节我们仅围绕释放内存这一主题,介绍必要的内存管理知识,然后改造【例 9.11.2】的程序,使其具备卸载和解驻功能,完成最终设计。

1. DOS 内存管理

DOS 把常规内存划分为若干个连续的内存块,并且在每个内存块的上方建立一个“内存控制块”(Memory Control Block, MCB)。从而形成一个“MCB 链”,DOS 通过内存控制块管理它属下的内存块。

每个 MCB 自身占用 16 个字节,前 5 个字节是有用的。

第 0 个字节:是一个标志单元。内容为 4DH,表示这个 MCB 管理的内存块不是最后的内存块;内容为 5AH,表示这个 MCB 管理的是最后一块内存;内容不为 4DH,不为 5AH,表明 MCB 键被“拆断了”,DOS 将立即发出:“Memory allocation error”的信息,然后使系统瘫痪。

第 1,2 字节:表明是哪个程序占用了内存块。第 1,2 字节为 0,表明这块内存是空闲的,如果被程序占用,就用该程序的 PSP 段基址填充。

第 3,4 字节:表明这个 MCB 所管理的内存块大小(不包括 MCB 本身的 16 个字节),单位为“节”,一节等于 16 个字节。

DOS 在装载可执行文件时,主动执行 INT 21H 的 48H 号子功能,为待装入的文件申请了两块内存。一块称为“环境变量区”,用来存放系统的环境变量(如: DOS 的路径提示符,装入文件的文件名等),一块称为“程序区”,用来存放装入的可执行文件,以及它的程序段前缀 PSP。文件装入后,DOS 根据环境变量区的大小,文件块的大小,以及它们占用的内存块段地址,再去修改相关 MCB 的第 0~第 4 字节。

图 9-25 形象地给出了一个 COM 文件的内存映像,从中得到 3 条极为重要的结论:

① COM 文件调入内存后,占用两块内存区,即环境变量区和程序区,程序区又包括 COM 文件目标代码和它的程序段前缀 PSP。

② 环境变量区的段基址,被 DOS 保存在 PSP 段,偏移地址为 2CH、2DH 的两个单元中。

③ PSP 段的段基址被装入 CS、DS、ES、SS 中。

了解了上述情况之后就会知道,要想释放程序占用的内存空间,就必须采取两项措施:

① 找到驻留程序环境变量区的段基址 XXXX,调用 INT 21H 的 49H 号功能,即可

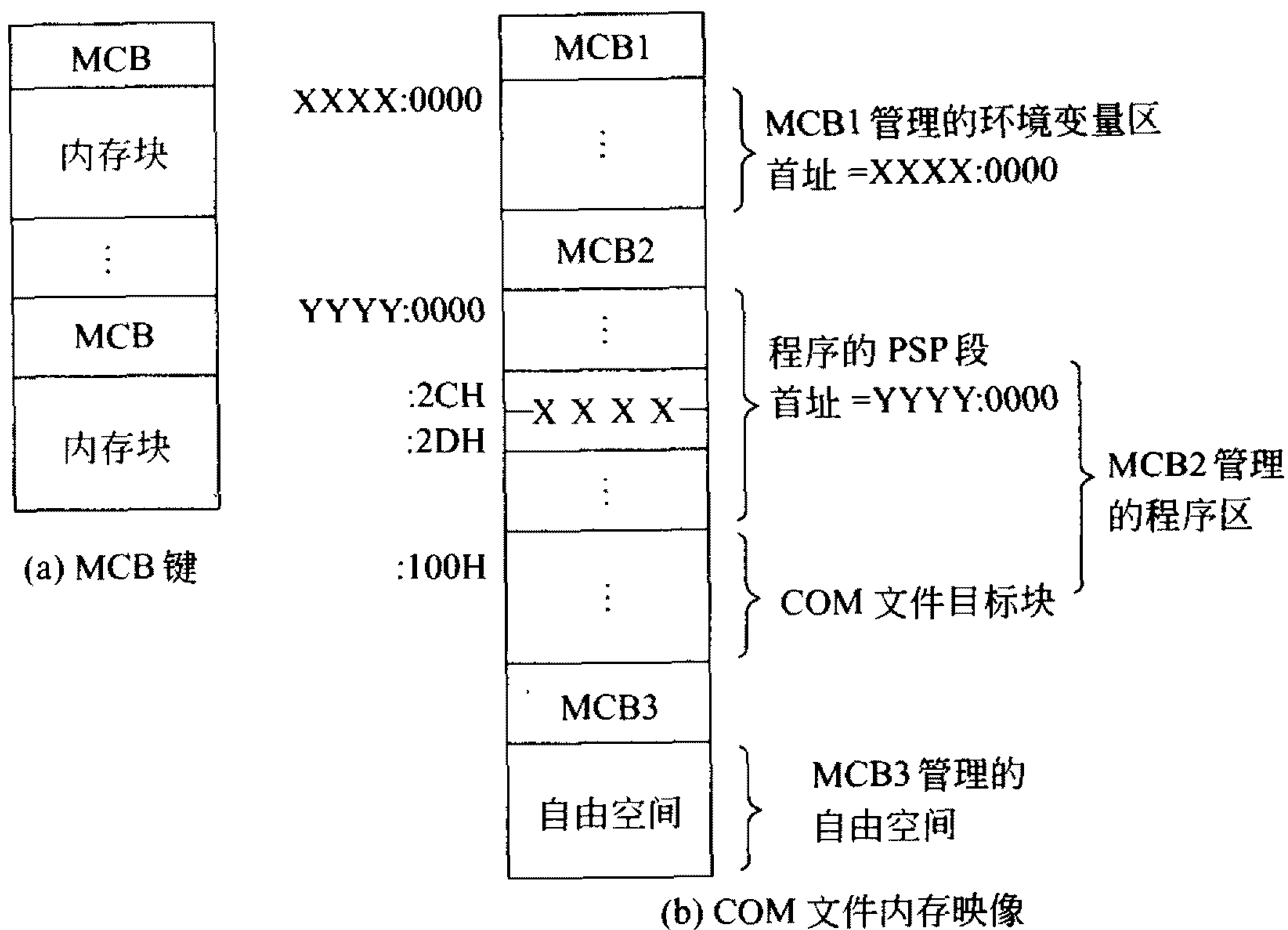


图 9-25 COM 文件内存映像

释放程序占用的环境变量区。

② 找到驻留程序 PSP 段的段基址 YYYYY,再次调用 INT 21H 的 49H 号功能,即可释放程序块占用的内存空间(包括 PSP 段)。

【INT 21H 的 49H 号子程序】

功能：释放由 48H 号子程序分配的内存块。

入口参数：AH=49H。

ES=待释放的内存块段基址。

出口参数：无。

到此为止,我们可以设计兼有解驻功能的驻留程序了。

2. 具有卸载和解驻功能的驻留程序

【例 9.11.3】 设计具有激活、卸载和解驻功能的驻留程序。

改造【例 9.11.2】程序,使之具有卸载和解驻功能。

【设计思路】

规定,程序驻留内存之后,每次按下“Ctrl+F11”系统就发出 3 声短促的音响,按下“Ctrl+F12”之后完成卸载和解驻功能,为此,例 9.11.2 的程序还需要采取两个措施,一是增加一个卸载和释放内存的程序段,二是增加一个热键识别,判断当前闭合键是否是“Ctrl+F12”,若是,则执行卸载和释放内存的程序段,完成卸载和解驻功能。

【程序清单】

```
                ;FILENAME: 9113. ASM
                . 486
CODE            SEGMENT USE16
                ASSUME  CS: CODE
```



```

                                ORG      100H
BEG:                            JMP      INIT
OLD09                          DD        ?           ;存放系统 9 型中断向量
OLD1C                          DD        ?           ;存放系统 1CH 型中断向量
HOTKEY                         DB        0           ;存放激活标志
COUNT                        DB        6
NEW09                          PROC
                                CLI
                                PUSHF
                                CALL     OLD09         ;转系统 9 型中断服务程序
                                PUSHA                ;保护现场
                                MOV      AH,11H
                                INT      16H
                                JZ       EXIT_1        ;无键入转
                                CMP      AX,8900H      ;Ctrl_F11?
                                JE       NEXT1         ;是,转
                                CMP      AX,8A00H      ;Ctrl_F12?
                                JNE      EXIT_1        ;不是,转
UNLOAD:                        MOV      AX, 2509H      ;恢复系统 9 型中断向量
                                MOV      DX,WORD PTR OLD09
                                MOV      DS,WORD PTR OLD09+2
                                INT      21H
                                MOV      AX,251CH      ;恢复系统 1CH 型中断向量
                                MOV      DX,WORD PTR OLD1C
                                MOV      DS,WORD PTR OLD1C+2
                                INT      21H
                                PUSH     ES
                                MOV      AX,CS
                                MOV      ES,AX          ;PSP 段基址存入 ES
                                MOV      AH,49H        ;释放程序块占用的内存空间
                                INT      21H          ;(包括 PSP 段)
                                MOV      ES,ES:[2CH]   ;环境变量区段基址存入 ES
                                MOV      AH,49H
                                INT      21H          ;释放环境变量区占用的内存空间
                                POP      ES
                                JMP      NEXT2
NEXT1:                          MOV      HOTKEY,-1     ;设置激活标志
NEXT2:                          MOV      AH,10H
                                INT      16H          ;清除热键信息
EXIT_1:                         POPA
                                IRET                  ;恢复现场
                                ;中断返回
NEW09                          ENDP
;-----
NEW1C                          PROC

```

```

                CMP     HOTKEY,-1           ;有激活标志吗?
                JNE     EXIT_2             ;没有转
ALARM:          IN      AL,61H
                XOR     AL,00000011B
                OUT     61H,AL             ;打开或者关闭扬声器
                DEC     COUNT             ;计数
                JNZ     EXIT_2             ;不满6次转
                MOV     COUNT,6
                MOV     HOTKEY,0           ;清除激活标志
                JMP     EXIT_2
EXIT_2:         IRET
NEW1C          ENDP
;-----
INIT:           MOV     AX,3509H           ;置换9型中断向量
                INT     21H
                MOV     WORD PTR OLD09,BX
                MOV     WORD PTR OLD09+2,ES
                MOV     AX,2509H
                MOV     DX,OFFSET NEW09
                INT     21H
                MOV     AX,351CH           ;置换1CH型中断向量
                INT     21H
                MOV     WORD PTR OLD1C,BX
                MOV     WORD PTR OLD1C+2,ES
                MOV     AX,251CH
                MOV     DX,OFFSET NEW1C
                INT     21H
                MOV     DX,OFFSET INIT
                INT     27H                 ;驻留退出
CODE           ENDS
                END     BEG

```

【程序调试】

上述程序的可执行文件(9113.COM)调入内存之后,每次按下“Ctrl+F11”能够发出3声短促的音响,这表明程序已经驻留并能正常激活,按下“Ctrl+F12”之后,再按下“Ctrl+F11”没有响声,这表明驻留程序已经被卸载,但是它所占用的内存空间是否被释放了呢? 还需要测试。

执行DOS程序MEM.EXE可以了解内存被占用的情况,上述的可执行文件调入内存驻留返回之后,在DOS提示符之下键入:

```
MEM/C |MORE
```

从屏显的内容可以查到驻留程序的文件名以及它占用的内存大小,按下“Ctrl+F12”之后,再次键入上述命令,从屏显的内容找不到刚才驻留的文件名,这说明程序已经被解

除驻留了。

习 题

1. 叙述可屏蔽中断处理的全过程。
2. 中断系统应具备哪些基本功能？
3. 什么是中断向量和中断向量表？中断类型码和中断向量的关系是什么？
4. 可屏蔽硬件中断的中断源是哪些？
5. CPU 响应可屏蔽中断的条件是什么？
6. CPU 响应非屏蔽中断的条件是什么？
7. 什么叫系统的保留中断和用户中断？在微型计算机系统上开发用户中断程序时应采取哪些措施？
8. 键盘硬中断和键盘软中断的关系是什么？
9. 用户怎样设计自己的键盘中断服务程序？用户键盘中断服务程序怎样和系统键盘中断服务程序衔接？

微型计算机系统串行通信

10.1 串行通信基础

通信的基本方式分为并行通信和串行通信。并行通信是指数据的所有位同时被传送,串行通信是指数据用一根传输线被逐位的顺序传送。串行通信又分为串行同步通信和串行异步通信。通常所说的串行通信指的是串行异步通信。

10.1.1 串行通信类型

1. 串行异步通信

串行异步通信是指一帧字符用起始位和停止位来完成收发同步。图 10-1 是串行异步通信的标准数据格式。

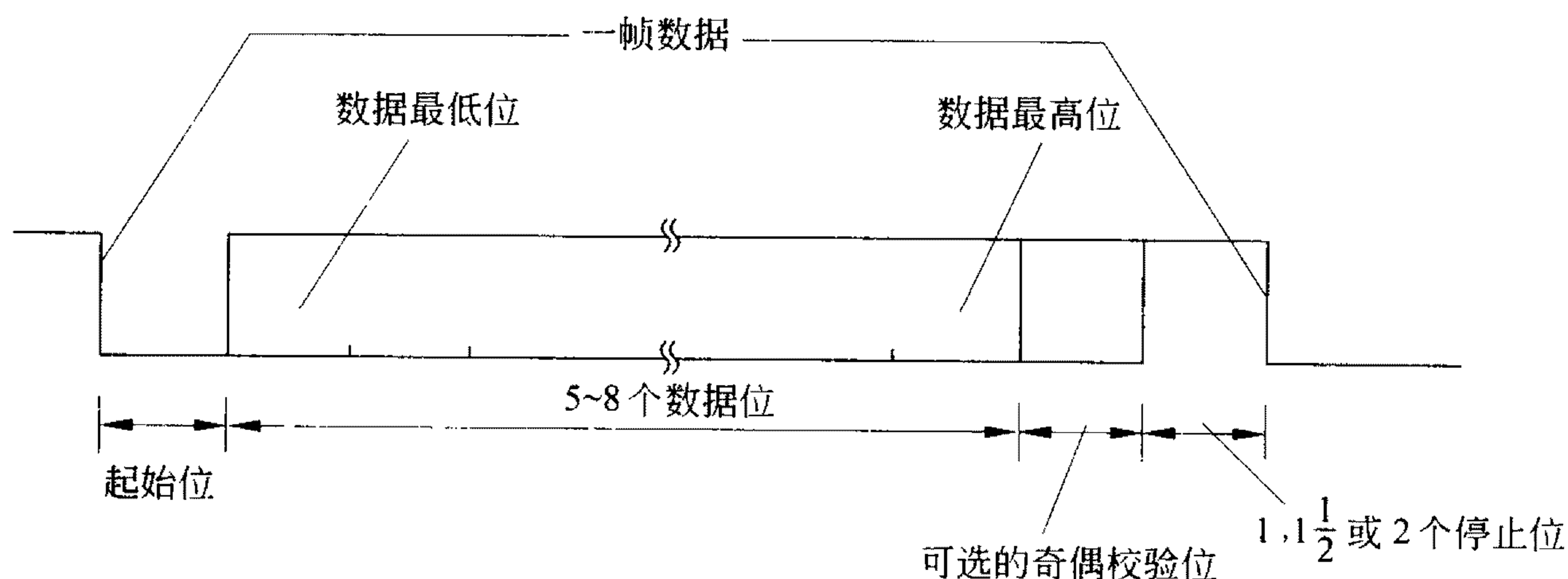


图 10-1 异步通信数据格式

如图 10-1 所示,异步通信时,一帧字符以起始位开始,然后是数据位和奇偶校验位,最后以停止位结束,起始位之后是数据的最低位。

传送开始时,接收设备不断检测传输线,当检测到一系列的“1”之后检测到一个“0”,便启动内部计数器开始计数。当计数到一个数据位宽度的一半时,又一次采样数据传输线,若其仍为低电平,则确认是一个起始位的到来,标志着一帧字符的开始,然后以位时间(1/波特率)为间隔,移位接收所规定的的数据位和奇偶校验位,拼装成一个字符的并行

字节,此后应接收到规定长度的停止位“1”,若没有收到,则设置“帧错误”标志。若校验有错,则设置“校验错”标志。只有既无帧出错又无奇偶校验错的接收数据才是正确的。一帧字符接收完毕,接收设备继续测试传输线,监测下一帧字符起始信号的到来。

异步通信是按字符传输的,接收设备在收到起始位信号后,只要在一个字符的传输时间内能和发送设备保持同步就能正确接收。若接收设备和发送设备两者的时钟略有偏差,字符之间的停止位和空闲位将为这种偏差提供一种缓冲,不会因累积效应而导致错位,接收端对异步通信每一个字符的起始位都重新校准时钟。

2. 串行同步通信

串行同步通信是采用同步字符来完成收发双方同步的。

异步通信由于要在每个字符前后附加起始位、停止位,有约 20% 的附加数据,传输效率不高。同步通信方式所用的数据格式没有起始位和停止位,一次传送的字符个数可以变化。在传送前,先按照一定的格式,将各种信息装配成一个数据包,该数据包包括一个或二个供接收方识别用的同步字符,其后紧跟着需传送的 n 个字符(n 的大小由用户设定且可变),最后是两个校验字符。同步通信的数据格式如图 10-2 所示。

同步字符	数据 1	2	3	...	n	校验字符 1	校验字符 2
------	------	---	---	-----	---	--------	--------

(a) 单同步数据格式

同步字符 1	同步字符 2	数据 1	2	...	n	校验字符 1	校验字符 2
--------	--------	------	---	-----	---	--------	--------

(b) 双同步数据格式

数据 1	2	3	4	...	n	校验字符 1	校验字符 2
------	---	---	---	-----	---	--------	--------

(c) 外同步数据格式

图 10-2 同步通信数据格式

接收设备首先搜索同步字符,在收到同步字符后,开始接收数据。在传输过程中,发送设备和接收设备要保持完全同步。如果因为某些原因,接收漏位,则其后的数据接收是错误的,这种错误可由校验字符查出。

在同步通信中要求使用同一时钟作为发送设备和接收设备的同步信号。在近距离通信时,可以在传输线中增加一根时钟信号线,用同一时钟发生器驱动收发设备;在远距离通信时,可以通过调制/解调器从数据流中提取同步信号,利用锁相技术可接收和发送时钟频率完全相同的接收时钟信号。

10.1.2 串行数据传输方式

串行数据传输方式有单工方式、半双工方式和全双工方式。

1. 单工方式

单工方式只允许数据按照一个固定的方向传送,如图 10-3 所示。

2. 半双工方式

半双工方式要求收发双方均具备接收和发送数据的能力,如图 10-3 所示,由于只有一条信道,数据不能在两个方向上同时传送。

3. 全双工方式

在全双工方式中,收发双方可以同时进行数据传送,如图 10-3 所示。

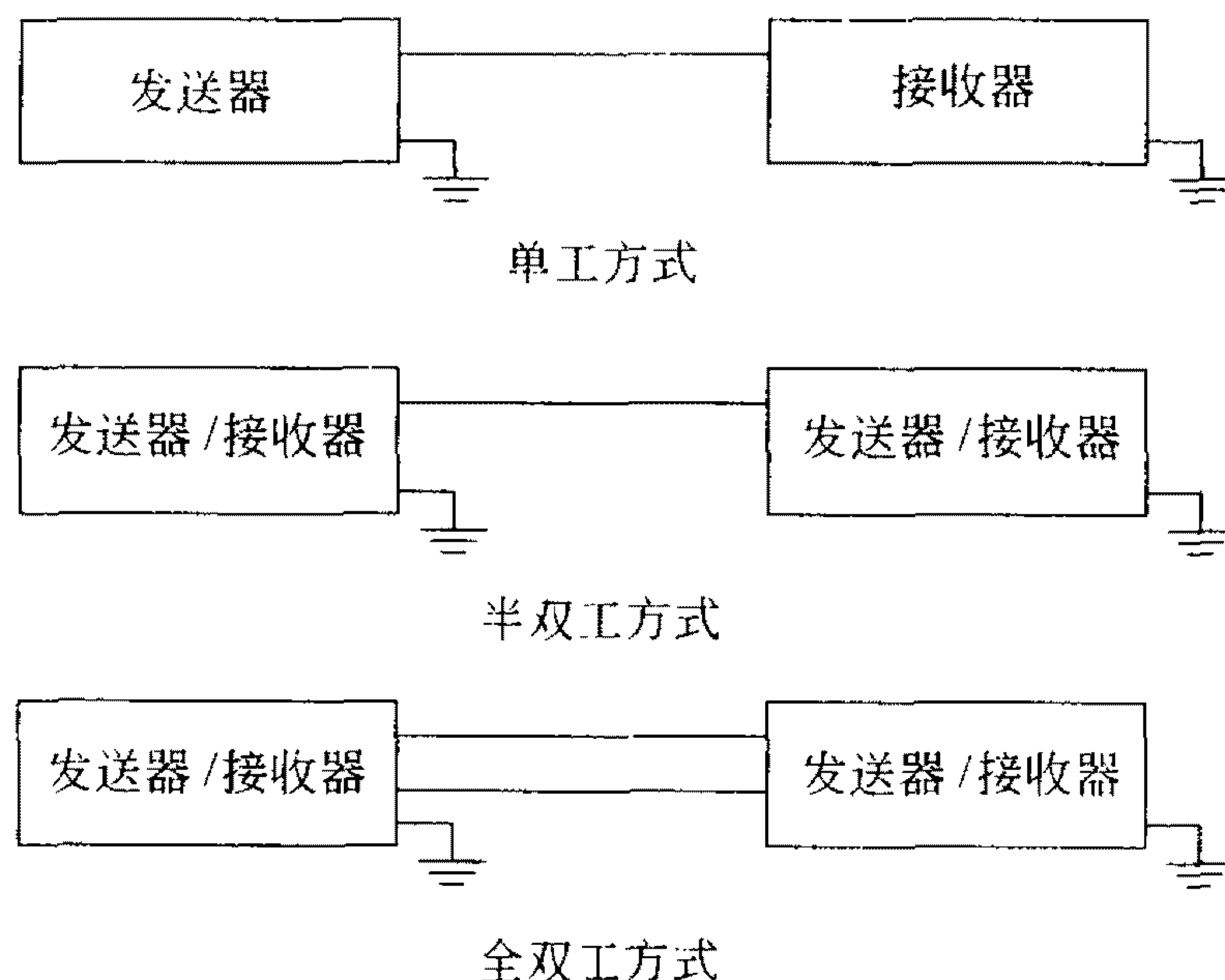


图 10-3 串行数据传送方式

10.1.3 串行异步通信协议

为了能够正常通信,发送方和接收方必须共同遵守一些通信协议,包括收发双方的同步方式、传输控制步骤、差错检验方式、数据编码、数据传输速率、通信报文的格式以及控制字符的定义等。

串行通信协议包括异步通信协议和同步通信协议。以下介绍串行异步通信协议。

1. 一帧数据的格式

一帧数据包括起始位、数据位、奇偶校验位和停止位四部分,收、发双方预置的帧数据格式必须一致。

(1) 起始位

传输线上若没有数据传输时,处于逻辑 1 状态。一帧字符开始,首先发送起始位,起始位是一位逻辑 0。接收设备检测到逻辑 0 信号后,开始接收数据。起始位的作用是使收发双方在传送数据位前协调同步。

(2) 数据位

起始位之后是数据位,数据位从最低位开始发送,数据位的个数为 5~8 位。

(3) 奇偶校验位

数据位发送完毕后,发送奇偶校验位。通信双方约定采用一致的奇偶校验方式,如

果是偶校验传输,那么数据位和奇偶校验位逻辑 1 的总个数应为偶数个;如果是奇校验传输,则数据位和奇偶校验位逻辑 1 的总个数应为奇数个;也可以进行无校验传输。

(4) 停止位

在奇偶校验位或数据位(当无奇偶校验时)之后发送的是停止位。停止位可以是 1 位、1.5 位或 2 位的逻辑 1 信号。

2. 通信速率

通信速率又称波特率,表示每秒钟传送 0、1 代码的位数(包括起始位、校验位、停止位),单位为“波特”。收发双方的通信速率必须一致。

3. 串行通信接口标准

在串行通信中,数据终端设备与数据通信设备之间的连接,要符合“接口标准”,目前在计算机通信中使用最广泛的是 RS-232C 标准。

RS-232C 标准是美国电子工业协会(EIA)在 1969 年公布的数据通信标准,它对信号的电平标准和控制信号的定义进行了规定。

(1) 控制信号的定义

PC 系列机通常有两个串行口:COM₁ 和 COM₂,使用 9 针或 25 针两种连接器,符合 RS-232C 接口标准。

对于 25 针连接器,其中 22 个引脚的功能均已定义,在计算机通信中常用的只有 9 个引脚,表 10-1 给出了计算机通信中常用的 RS-232C 信号标准。

表 10-1 计算机通信中常用的 RS-232C 信号标准

9 针连接器端子号	25 针连接器端子号	名称	方向	功 能
3	2	TXD	输出	发送数据(Transmit Data)
2	3	RXD	输入	接收数据(Receive Data)
7	4	$\overline{\text{RTS}}$	输出	请求发送(Request To Send)
8	5	$\overline{\text{CTS}}$	输入	允许发送(Clear To Send)
6	6	$\overline{\text{DSR}}$	输入	数据设备准备好(Data Set Ready)
5	7	GND		信号地(Signal Ground)
1	8	$\overline{\text{DCD}}$	输入	载波检测(Carrier Detect)
4	20	$\overline{\text{DTR}}$	输出	数据终端准备好(Data Term Ready)
9	22	RI	输入	振铃指示(Ring Indicator)

(2) 信号电平标准

RS-232C 采用负逻辑。规定逻辑“1”为 $-3\text{V} \sim -15\text{V}$,规定逻辑“0”为 $+3\text{V} \sim +15\text{V}$ 。

当计算机与外设进行通信时,由于 TTL 电平是正逻辑,因此必须有相应的电平转换电路。通常采用的是 MC1488 和 MC1489 电平转换器。图 10-4 是 MC1488 和 MC1489 的逻辑图。

MC1488 可接收 TTL 电平,输出 RS-232C 电平。MC1489 可输入 RS-232C 电平,输

出 TTL 电平。

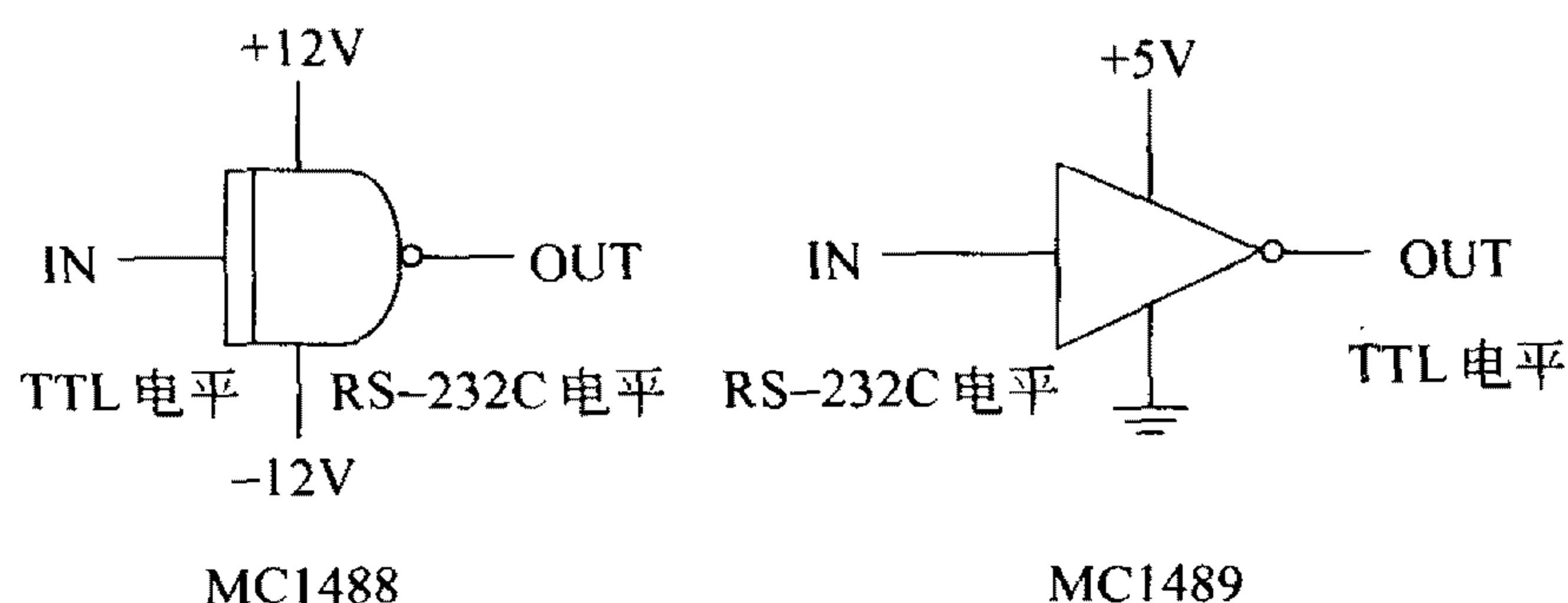


图 10-4 MC1488 和 MC1489 逻辑图

4. 信号的调制和解调

在串行通信中,数据终端一般采用计算机,数据要通过数据通信设备传送,数据通信设备一般指调制解调器。

由于计算机经 RS-232C 接口输出的是数字信号,要求传输线的频带较宽。在远程数据通信时,通信线路大多利用电话线,由于频带不宽,传送数字信号会产生失真,但传送模拟信号,则失真较小。因此在远距离通信时,发送方要用调制器把数字信号调制为模拟信号,接收方要用解调器进行解调,将模拟信号转换成数字信号。如图 10-5 所示。

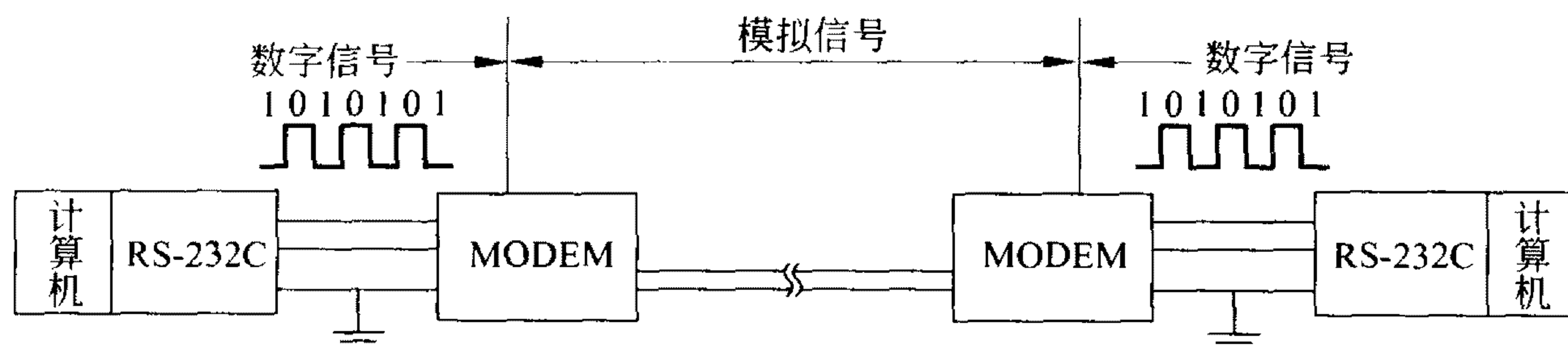


图 10-5 调制与解调示意图

多数情况下,通信是双向的,调制器和解调器设计在一个装置中,称为调制解调器(MODEM)。调制解调器的类型比较多,有振幅键控(ASK)、频移键控(FSK)和相移键控(PSK)。

实现串行通信有专用的接口芯片。常用的有 USART(通用同步/异步接收/发送器),如 Intel 8251;UART(通用异步接收/发送器),如 Ins 8250。

无论是 UART,还是 USART,均能实现数据发送时所需要的并→串转换以及数据被 CPU 接收时所需的串→并转换。

10.2 可编程串行异步通信接口芯片 8250

Ins 8250 是可编程串行异步通信接口芯片,有 40 条引脚,双列直插式封装,使用单一的 +5V 电源,能实现数据的串→并及并→串转换,支持异步通信协议。片内有时钟产生电路,波特率可变。对外有调制解调器控制信号,可直接与 MODEM 相连。高档微型计算机中使用多功能芯片,但其串行接口的功能与 8250 兼容。

10.2.1 8250 的内部结构

8250 内部包括数据总线缓冲器、选择和读/写控制逻辑、发送器、接收器、调制解调控制电路、通信线控制寄存器、通信线状态寄存器、波特率发生控制电路和中断控制逻辑。图 10-6 是 8250 内部结构图。

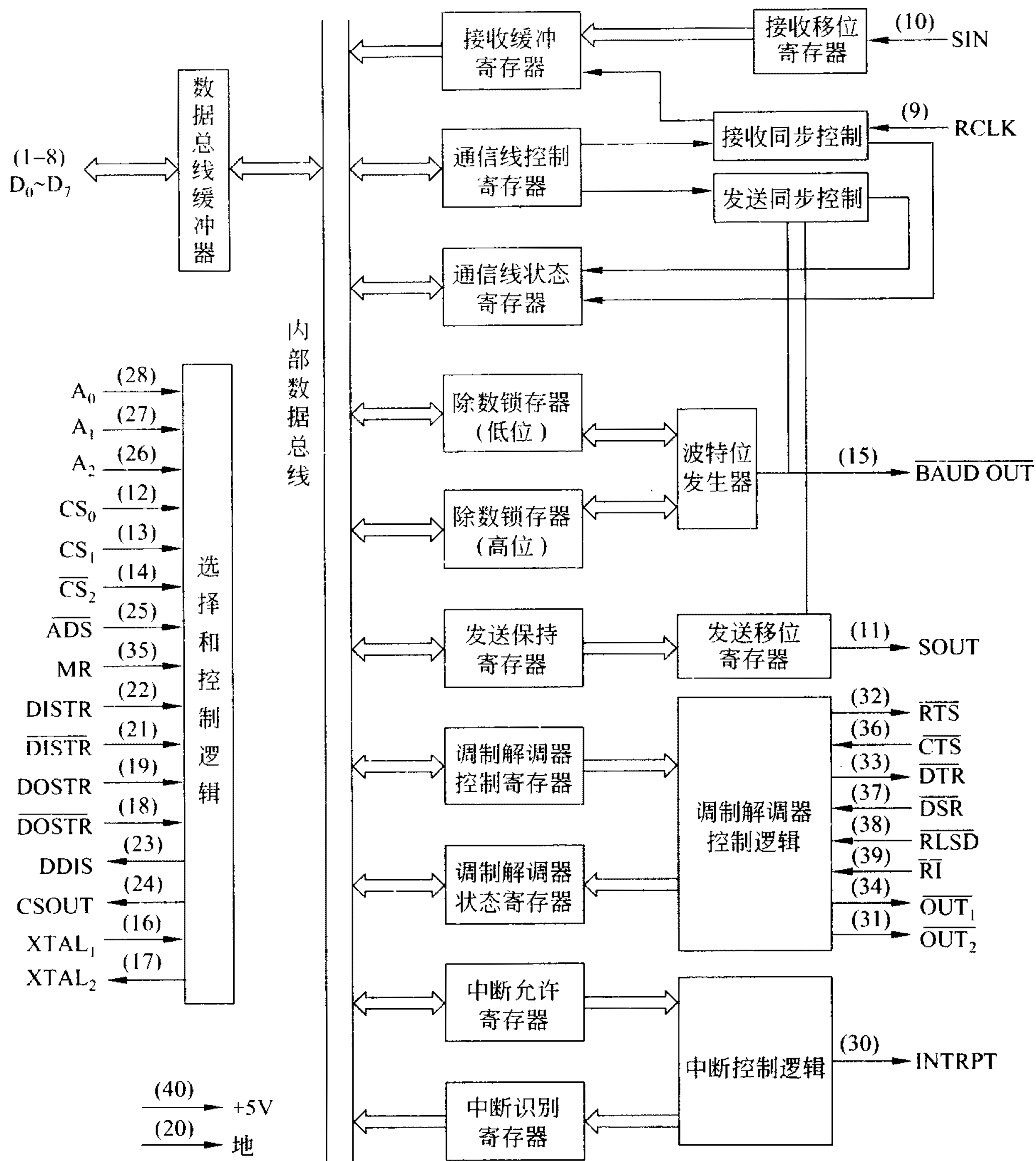


图 10-6 8250 内部结构图

1. 数据总线缓冲器

数据总线缓冲器是 8250 与 CPU 之间的数据通道,来自 CPU 的各种控制命令以及待发送的字符信息经该通道到达 8250 内部;8250 内部的状态信息、数据信息经该通道送至系统数据线。

2. 选择和读/写控制逻辑

接收来自 CPU 的各种控制信息,从而确定操作方式。

3. 发送器

它由发送保持寄存器、发送移位寄存器和发送同步控制三部分组成,待发送的数据写入发送保持寄存器。数据发送时,发送保持寄存器的内容自动转存到发送移位寄存器,在发送器时钟的控制下,发送移位寄存器自动添加起始位、校验位和停止位,将并行数据转换成串行数据,经 SOUT 引脚发送出去。

4. 接收器

它由接收移位寄存器、接收缓冲寄存器和接收同步控制三部分组成。在接收器时钟的控制下,来自引脚 SIN 的串行数据被逐位存入接收移位寄存器,在移位过程中自动进行校验,并去掉起始位、停止位和校验位,然后将转换后的并行数据存入接收缓冲寄存器,等待 CPU 读取。

5. 调制/解调控制电路

8250 内部的调制/解调控制电路提供一组通用的控制信号,使 8250 可直接与调制解调器相连,以完成远程通信任务。

6. 通信线控制寄存器和通信线状态寄存器

通信线控制寄存器指定串行通信的数据格式,通信线状态寄存器提供串行数据发送和接收时的状态,供 CPU 读取和处理。

7. 波特率发生控制电路

它由波特率发生器、存放分频系数低位和高位字节的除数寄存器组成。

8250 使用频率为 1.8432MHz 的基准时钟输入信号,通过内部分频产生发送器时钟和接收器时钟,发送器时钟和接收器时钟的频率是数据传输波特率的 16 倍。

$$16 \times \text{波特率} = 1843200 \div \text{分频系数} (\text{分频系数即为除数})$$

8. 中断控制逻辑

它由中断允许寄存器、中断识别寄存器和中断控制逻辑三部分组成,对中断优先权、中断申请等进行管理。

10.2.2 8250 的引脚功能

8250 共有 40 个引脚,如图 10-7 所示。

$D_7 \sim D_0$ 为 8 根并行数据线,其他信号线如下:

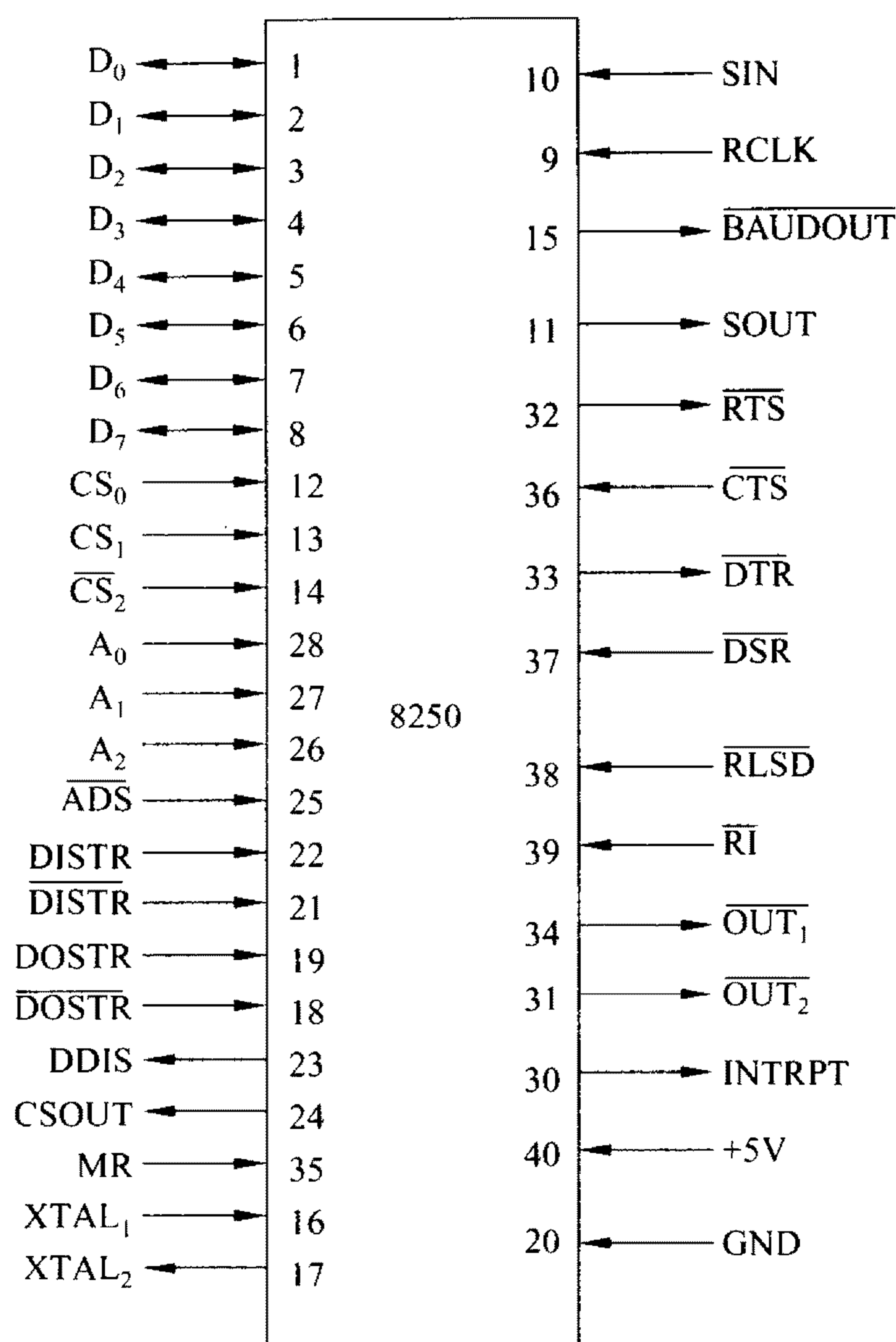


图 10-7 8250 引脚图

1. 地址控制信号

当片选信号 $CS_0 = 1$ 、 $CS_1 = 1$ 、 $\overline{CS_2} = 0$ 时, CPU 可访问 8250, 由 $A_2 \sim A_0$ 决定所访问的内部寄存器。当地址选通信号 \overline{ADS} 为低电平时, 锁存片选信号 (CS_0 、 CS_1 、 $\overline{CS_2}$) 及 $A_2 \sim A_0$ 的地址信号, 保证读写操作期间的地址稳定, 直到 \overline{ADS} 变为高电平, 这些地址选择信号才允许变化。如果确认在对芯片进行读写时, 不会出现地址不稳定现象, 可不必锁存, 而将 \overline{ADS} 输入脚接地。引脚 CSOUT 是芯片被选中的输出指示信号, 当 8250 被 CS_0 、 CS_1 、 $\overline{CS_2}$ 信号选中时, CSOUT 为高电平, 表明可以进行数据传送, 这一信号通常因不需要而悬空。

2. 读/写控制信号

8250 的读/写控制信号有两对, 每一对信号作用相同, 但有效电平不同。当 8250 被选中时, 数据输入选通信号 DISTR (高电平有效) 和 \overline{DISTR} (低电平有效) 中有一个信号有效, CPU 从被选择的内部寄存器中读出数据; 而数据输出选通信号 DOSTR (高电平有效) 或 \overline{DOSTR} (低电平有效) 有效时, CPU 将数据写入被选择的寄存器。

若选择 \overline{DISTR} 接 CPU 的 \overline{IOR} , 则应将 DISTR 接地, 变为无效。

若选择 DOSTR 接 CPU 的 \overline{IOW} , 则应将 \overline{DOSTR} 接高电平。

DDIR 是禁止驱动器输出信号。当 CPU 从 8250 读取数据时, DDIR 为低电平。当 DDIR 为高电平时, 用来禁止外部收发器对系统总线的驱动。

3. 中断控制和复位控制信号

8250 有 4 个内部中断源, 分别是接收错中断、接收中断、发送中断和调制解调器中断。

8250 本身具有很强的中断控制和优先权判决处理能力, 它的中断请求引脚 INTRPT 在满足一定条件下(当接收数据错, 接收数据就绪, 发送保持寄存器空, 调制解调器状态改变, 并且芯片内的中断允许寄存器相应位置 1 时)变成高电平, 产生中断请求。在调制解调器控制寄存器中, $\overline{OUT_1}$ 和 $\overline{OUT_2}$ 是由用户通过编程使其有效的两个输出引脚。在 PC 系列机中, $\overline{OUT_1}$ 未用, $\overline{OUT_2}$ 用来作为中断请求信号 INTRPT 的输出控制, 如图 10-8 所示。

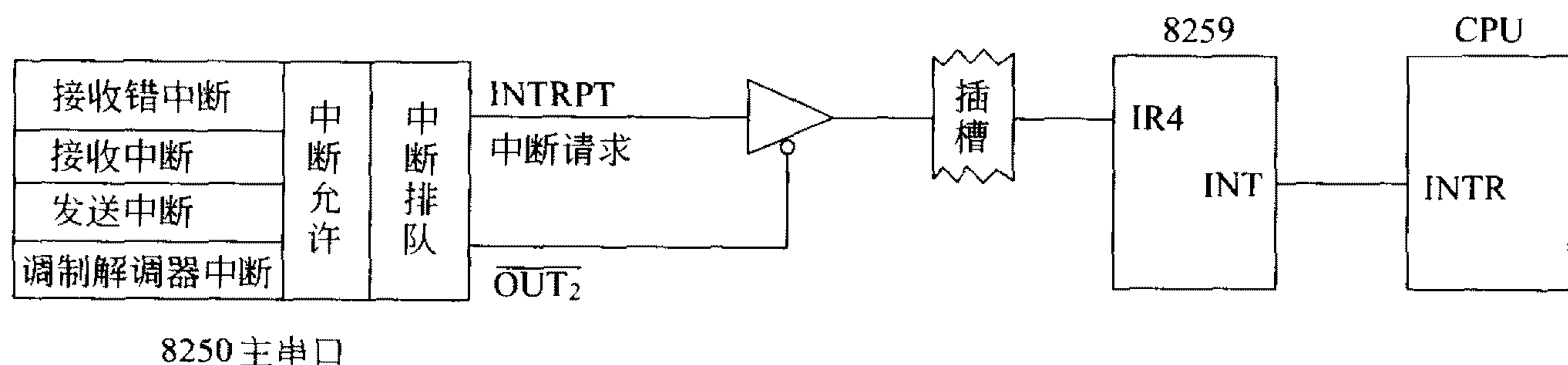


图 10-8 8250 中断请求信号与 CPU 的连接示意图

当系统复位时, RESET 信号送到 8250 的主复位端 MR, 当 MR=1 时, 8250 进入复位状态。

4. 时钟与传送速率控制信号

外部晶体振荡电路产生的 1.8432MHz 信号送到 8250 的 XTAL₁ 端, 作为 8250 的基准工作时钟。XTAL₂ 引脚是基准时钟信号的输出端, 可用作其他功能的定时控制。

外部输入的基准时钟, 经 8250 内部波特率发生器(分频器)分频后产生发送器时钟, 并经 BAUDOUT 引脚输出。8250 的接收器时钟引脚 RCLK 可接收由外部提供的接收器时钟信号。若采用 8250 内部的发送器时钟作为接收时钟, 则只要将 RCLK 引脚和 BAUDOUT 引脚直接相连。在 PC 系列机上, 用 8250 进行数据传送, 发送波特率和接收波特率是一致的。

5. 和外设/调制解调器之间的联络信号

共有 6 个联络信号 \overline{RTS} 、 \overline{CTS} 、 \overline{DSR} 、 \overline{DTR} 、 \overline{RLSD} 、 \overline{RI} 和 2 根串行数据信号线 SOUT、SIN。

\overline{RTS} : 请求发送。 \overline{RTS} 由 8250 输出, $\overline{RTS}=0$, 表示 8250 通知外设或调制解调器, 8250 发送准备完毕, 请对方做好接收准备。

$\overline{\text{CTS}}$: 发送允许。 $\overline{\text{CTS}}$ 由外设或调制解调器送往 8250,是对 $\overline{\text{RTS}}$ 信号的响应, $\overline{\text{CTS}}=0$,表示外设或调制解调器接收准备完毕,8250 可以发送数据。

$\overline{\text{DTR}}$: 数据终端准备好。 $\overline{\text{DTR}}$ 由 8250 输出, $\overline{\text{DTR}}=0$,表示 8250 通知外设或调制解调器,8250 接收准备完毕,对方可以发送数据了。

$\overline{\text{DSR}}$: 数据设备准备好。 $\overline{\text{DSR}}$ 由外设或调制解调器送往 8250,是对 8250 发出的 $\overline{\text{DTR}}$ 信号的响应, $\overline{\text{DSR}}=0$,表示外设或调制解调器发送准备完毕。

$\overline{\text{RLSD}}$: $\overline{\text{RLSD}}=0$,表示调制解调器已接收到数据载波,8250 应立即开始接收解调后的数据。

$\overline{\text{RI}}$: 振铃指示输入信号。当 $\overline{\text{RI}}=0$ 时,表示调制解调器接收到电话线上的振铃信号。

10.2.3 8250 内部寄存器

8250 内部有 10 个可寻址的寄存器,分为三组:第一组用于实现数据传输,有发送保持寄存器、接收缓冲寄存器;第二组用于工作方式、通信参数的设置,称为控制字寄存器,有通信线控制寄存器、除数寄存器、调制解调器控制寄存器、中断允许寄存器;第三组称为状态寄存器,有通信线状态寄存器、调制解调器状态寄存器和中断识别寄存器。

表 10-2 给出了微型计算机系统 8250 内部寄存器的端口地址。

表 10-2 微型计算机系统 8250 内部寄存器端口地址

DLAB	A ₂	A ₁	A ₀	被访问的寄存器	主串口地址	辅串口地址
0	0	0	0	接收缓冲器(读) 发送保持寄存器(写)	3F8 H	2F8 H
0	0	0	1	中断允许寄存器	3F9 H	2F9 H
×	0	1	0	中断识别寄存器	3FA H	2FA H
×	0	1	1	通信线控制寄存器	3FB H	2FB H
×	1	0	0	调制解调器控制寄存器	3FC H	2FC H
×	1	0	1	通信线状态寄存器	3FD H	2FD H
×	1	1	0	调制解调器状态寄存器	3FE H	2FE H
1	0	0	0	除数寄存器(低字节)	3F8 H	2F8 H
1	0	0	1	除数寄存器(高字节)	3F9 H	2F9 H

备注:DLAB 不是芯片引脚,是通信线控制寄存器的 D₇ 位(寻址位)

1. 发送保持寄存器(3F8H/2F8H)

该寄存器保存 CPU 送出的并行数据,转移至发送移位寄存器。发送移位寄存器在发送器时钟的作用下,将并行数据按设定的帧格式添加起始位、校验位和停止位,转换成串行数据,从 SOUT 引脚输出。只有在发送保持寄存器空闲时,CPU 才能写入新数据。

2. 接收缓冲寄存器(3F8H/2F8H)

外部的串行数据在接收器时钟作用下,从 SIN 引脚输入到接收移位寄存器,去掉起始位、校验位和停止位,转换成并行数据,转换后的并行数据存入接收缓冲寄存器,等待

CPU 读取。

3. 通信线状态寄存器(3FDH/2FDH)

该寄存器提供数据传输的状态信息,其各位含义如下:

D₀ 位: 接收数据准备好(接收缓冲器满)标志位。D₀=1,表示接收器已接收到一帧完整的数据,并已转换成并行数据,存入接收缓冲寄存器。

D₁ 位: 溢出错标志位。D₁=1,表示接收缓冲器中的字符未取走,8250 又接收到新输入的数据,造成前一数据被破坏。

D₂ 位: 奇偶错标志位。D₂=1,表示接收到的数据有奇偶错。

D₃ 位: 帧错(接收格式错)标志位。D₃=1,表示接收到的数据没有正确的停止位。

D₄ 位: 线路间断标志位。D₄=1,表示收到长时间“0”信号(即中止信号)。

以上 D₁~D₄ 均为错误标志,只要其中有一位为 1,在中断允许的情况下,8250 内部将产生“接收数据错”中断,当 CPU 读取状态寄存器后,自动复位。

D₅ 位: 发送保持寄存器空闲标志位。D₅=1,表示数据已从发送保持寄存器转移到发送移位寄存器,发送保持寄存器空闲,CPU 可以写入新数据。当新数据送入发送保持寄存器后,D₅ 为 0。

D₆ 位: 发送移位寄存器空闲标志位。D₆=1,表示一帧数据已发送完毕。当新的数据由发送保持寄存器移入发送移位寄存器时,该位为 0。

D₇ 位: 恒为 0。

D₀ 位(接收缓冲器满)和 D₅ 位(发送保持寄存器空闲)是串行接口最基本的标志位,它们决定了 CPU 能否对 8250 进行读写操作。只有当 D₀=1 时,CPU 才能读数据;只有当 D₅=1 或 D₆=1 时,CPU 才能写数据。

以下是 CPU 查询通信线状态寄存器进行接收和发送数据的程序段:

```
SCAN:      MOV      DX,3FDH
            IN        AL,DX          ;读取通信线状态字
            TEST     AL,00011110B   ;检查有无错误标志
            JNZ      ERR             ;有错,转出错处理
            TEST     AL,01H         ;无错,检查接收数据是否准备好?
            JNZ      RECEIVE        ;准备好,转接收程序
            TEST     AL,20H         ;未准备好,检查发送保持寄存器空?
            JNZ      TRAS           ;已空,转发送程序
            JMP      SCAN           ;不空,循环等待

ERR:        ...
RECEIVE:    ...
TRAS:       ...
```

4. 中断允许寄存器(3F9H/2F9H)

该寄存器的 D₇~D₄ 位恒为 0。D₃~D₀ 位表示 8250 的 4 级中断是否被允许。D₀=1,允许接收到一帧数据后,内部提出“接收中断请求”。

$D_1=1$,允许发送保持寄存器空闲时,内部提出“发送中断请求”。

$D_2=1$,允许接收出错时,内部提出“接收数据错中断请求”。

$D_3=1$,允许调制解调器状态改变时,内部提出“调制解调器中断请求”。

8250 的 4 级中断,以“接收数据错中断”优先级最高,其次是“接收中断”、“发送中断”,“调制解调器中断”优先级最低。

5. 中断识别寄存器(3FAH/2FAH)

由于 8250 只能向 CPU 发出一个中断请求信号,为了识别是 8250 内部哪一个中断源引起的中断,在进入中断服务程序后,先读取中断识别寄存器的内容进行判断,然后再转入相应的处理程序。中断识别寄存器的 $D_7 \sim D_3$ 位恒为 0, D_0 位表示有无中断待处理, $D_0=1$,表示无中断待处理; $D_0=0$,表示有中断待处理。 $D_2 \sim D_1$ 位表示 4 种中断源识别码,其代表的中断源如表 10-3 所示。

表 10-3 8250 的中断源

中断识别寄存器 D_2 D_1 D_0	优先级	中 断 类 型	中断复位控制
0 0 1	—	无中断	
1 1 0	↓	接收数据箱	读通信线状态寄存器即可复位
1 0 0		接收数据准备好	读接收缓冲器可复位
0 1 0		发送保持寄存器空	写发送保持寄存器可复位
0 0 0		调制解调器状态改变	读调制解调器状态寄存器可复位

6. 调制解调器控制寄存器(3FCH/2FCH)

调制解调器控制寄存器是一个 8 位寄存器, $D_0 \sim D_3$ 位的状态直接控制相关引脚的输出电平。

D_0 位=1,使引脚 $\overline{DTR}=0$,从而使 RS-232C 引脚 \overline{DTR} 为 0。

D_1 位=1,使引脚 $\overline{RTS}=0$,从而使 RS-232C 引脚 \overline{RTS} 为 0。

D_2 位=1,使引脚 $\overline{OUT_1}=0$,PC 系列机未使用。

D_3 位=1,使引脚 $\overline{OUT_2}=0$,8250 能送出中断请求。

D_4 位通常置 0,设置 8250 工作在正常收/发方式;若 D_4 位置 1,则 8250 工作在内部自环方式,即发送移位寄存器的输出在芯片内部被回送到接收移位寄存器的输入。利用这个特点,可以编写程序测试 8250 的工作是否正常,而不须任何附加装置。

$D_7 \sim D_5$ 位恒为 0。

7. 除数寄存器(高 8 位 3F9H/2F9H,低 8 位 3F8H/2F8H)

除数寄存器为 16 位,由高 8 位寄存器和低 8 位寄存器组成。

8250 对 1.8432MHz 的时钟输入,采用分频的方法产生所要求的发送器时钟信号和

接收器时钟信号,分频系数由程序员分两次写入除数寄存器的高 8 位和低 8 位,除数(即分频系数)的计算公式如下:

$$\text{除数} = 1843200 \div (\text{波特率} \times 16)$$

表 10-4 列出了获得 14 种波特率所设置的除数寄存器值。

表 10-4 波特率与分频系数(即除数)对照表

波特率	除数高 8 位	除数低 8 位	波特率	除数高 8 位	除数低 8 位
50	09H	00H	1800	00H	40H
75	06H	00H	2000	00H	3AH
110	04H	17H	2400	00H	30H
150	03H	00H	3600	00H	20H
300	01H	80H	4800	00H	18H
600	00H	C0H	7200	00H	10H
1200	00H	60H	9600	00H	0CH

除数寄存器的值必须在 8250 初始化时预置。因此,必须先把通信线控制寄存器的最高位(DLAB)置为 1,然后分两次将除数写入高 8 位除数寄存器和低 8 位除数寄存器。

8. 通信线控制寄存器(3FBH/2FBH)

该寄存器规定串行异步通信的数据格式,如图 10-9 所示。

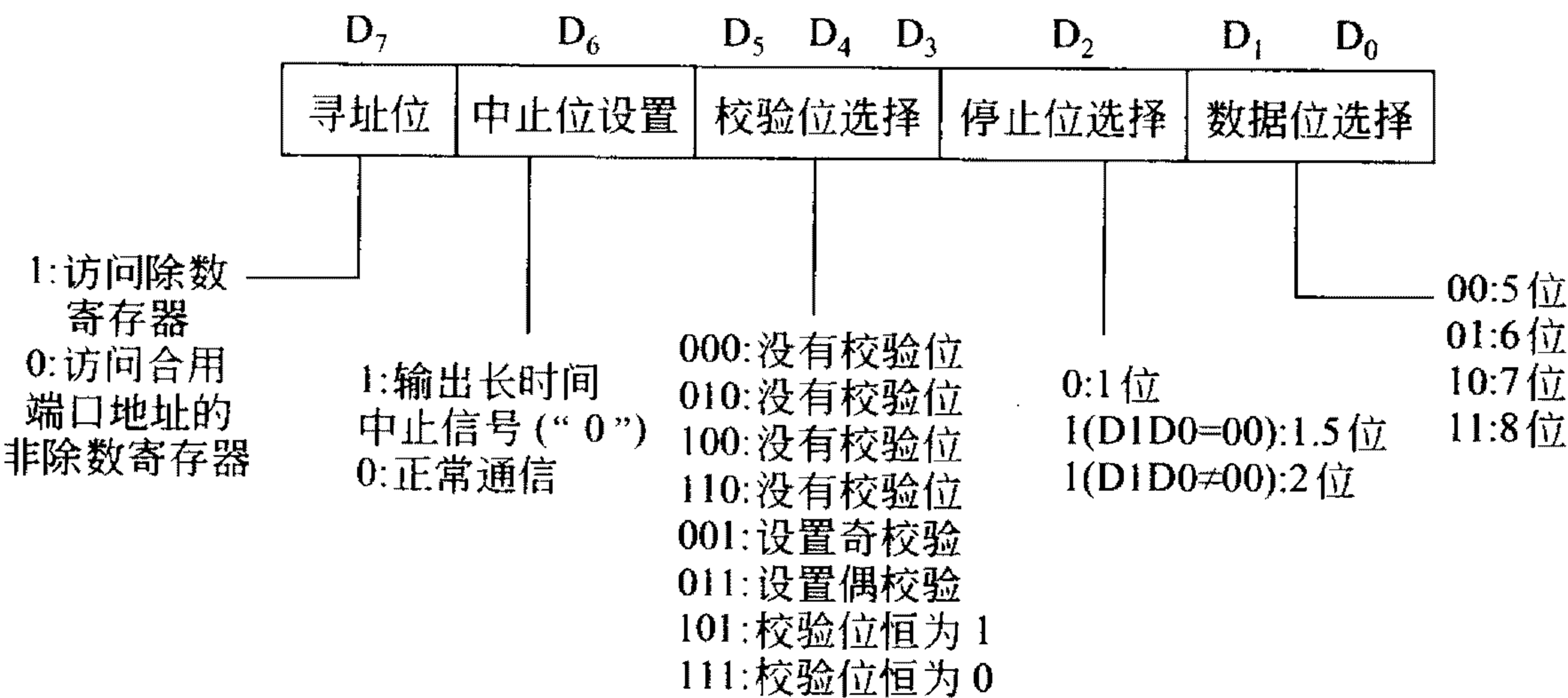


图 10-9 通信线控制寄存器格式

- D₀ 位和 D₁ 位: 规定一帧数据中数据位的位数。
- D₂ 位: 规定一帧数据中停止位的位数。
- D₃ 位、D₄ 位、D₅ 位: 规定一帧数据中的奇偶校验方式。
- D₆ 位: D₆ = 1, 8250 输出长时间中止信号。
- D₇ 位: 寻址位(DLAB), D₇ = 1, 访问除数寄存器。

8250 内部有 10 个可访问的 8 位寄存器,但 8250 只有 3 根用于端口选择的地址线(A₂、A₁、A₀),因此必然有某些寄存器合用一个端口地址。

8250 规定: 高 8 位除数寄存器和中断允许寄存器,二者合用一个端口地址;低 8 位除数寄存器、发送保持寄存器和接收缓冲寄存器三者合用一个端口地址。为了区别写入

合用端口的数据,8250 规定:通信线控制寄存器的 D_7 位为寻址位(DLAB),当 D_7 位=1 时,送往合用端口的数据将写入除数寄存器; D_7 位=0 时,送往合用端口的数据将写入非除数寄存器。

9. 调制解调器状态寄存器(3FEH/2FEH)

该寄存器反映 8250 与通信设备(如调制解调器)之间联络信号的当前状态以及变化情况。

$D_7 \sim D_4$ 记录了 4 个输入引脚的状态电平:

$D_7=1$ 表示输入引脚 $\overline{\text{RLSD}}=0$,调制解调器收到来自电话线的载波信号。

$D_6=1$ 表示输入引脚 $\overline{\text{RI}}=0$,调制解调器收到振铃信号。

$D_5=1$ 表示输入引脚 $\overline{\text{DSR}}=0$,调制解调器做好了发送准备,请 8250 准备接收。

$D_4=1$ 表示输入引脚 $\overline{\text{CTS}}=0$,调制解调器做好了接收准备,8250 可以发送数据。

$D_3 \sim D_0$ 记录了上一次读取该寄存器后,上述引脚是否发生过电平变化。

$D_3=1$ 表示输入引脚 $\overline{\text{RLSD}}$ 有电平变化。

$D_2=1$ 表示输入引脚 $\overline{\text{RI}}$ 有电平变化。

$D_1=1$ 表示输入引脚 $\overline{\text{DSR}}$ 有电平变化。

$D_0=1$ 表示输入引脚 $\overline{\text{CTS}}$ 有电平变化。

10.2.4 8250 的初始化编程

如果采用直接对端口操作的方式,8250 的初始化编程步骤为:

- ① 设置寻址位: $80\text{H} \rightarrow$ 通信线控制寄存器,使寻址位为 1。
- ② 将除数高 8 位/低 8 位 \rightarrow 除数寄存器高 8 位/低 8 位,确定通信速率。
- ③ 将 $D_7=0$ 的控制字写入通信线控制寄存器,规定一帧数据的格式。
- ④ 设置中断允许控制字:

若采用查询方式,则置中断允许控制字为 0。

若采用中断方式,则置中断允许寄存器的相应位为 1。

- ⑤ 设置调制解调器控制寄存器。

中断方式: $D_3=1$,允许 8250 送出中断请求信号。

查询方式: $D_3=0$ 。

内环自检: $D_4=1$ 。

正常通信: $D_4=0$ 。

【例 10.2.1】 编写子程序,采用直接对端口操作的方式对 PC 系列机主串口进行初始化。要求:

- ① 通信速率=1200 波特,一帧数据包括: 8 个数据位,1 个停止位,无校验。
- ② 采用查询方式,完成内环自检。

初始化子程序如下:

```
I8250    PROC
          MOV     DX,3FBH
          MOV     AL,80H
          OUT     DX,AL
          MOV     DX,3F9H
          MOV     AL,0
          OUT     DX,AL
          MOV     DX,3F8H
          MOV     AL,60H
          OUT     DX,AL
          MOV     DX,3FBH
          MOV     AL,03H
          OUT     DX,AL
          MOV     DX,3F9H
          MOV     AL,0
          OUT     DX,AL
          MOV     DX,3FCH
          MOV     AL,10H
          OUT     DX,AL
          RET
I8250    ENDP
```

10.3 串行通信程序设计

微型计算机系统有两个串行口,主串口 COM1(端口地址为 3F8H),辅串口 COM2(端口地址为 2F8H),结构相同。串口适配器组装在一块多功能卡上面(586 以上微型机串口适配器在主板上),多功能卡插在主板插槽中,通过总线与系统连接,串口用 25 芯或 9 芯连接器与外设备进行串行通信。

系统机串行口的核心器件是 I8250。串行通信的程序设计类型有单端自发自收(目的是测试串行口的好坏)、点到点双机通信。从交换方式上讲,CPU 与 I8250 之间可以用查询方式,也可用中断方式传送信息。从通信方式上讲,可以进行单工、半双工或全双工通信。从编程技巧上讲,可以对端口直接操作,也可以用 BIOS 通信软件完成数据发送和接收。

10.3.1 BIOS 通信软件

BIOS 通过 INT 14H 向用户提供了 4 个中断子程序,分别完成:串口初始化编程、发送一帧数据、接收一帧数据、测试通信线状态。

1. 串口初始化

【INT 14H 0 号子功能】 串口初始化。

入口参数:AH=0 串口初始化。

AL= 初始化参数。

DX=0 对主串口初始化,DX=1 对辅串口初始化。

出口参数: AH=通信线状态寄存器内容。

AL=MODEM 状态寄存器内容。

(1) 初始化参数的数据格式

初始化参数是一个 8 位的数据,它分为 4 个域:

① $D_7 D_6 D_5$ 用来选择波特率。

$D_7 D_6 D_5 = 000 \quad 001 \quad 010 \quad 011 \quad 100 \quad 101 \quad 110 \quad 111$

波特率 = 110 150 300 600 1200 2400 4800 9600

② $D_4 D_3$ 校验位选择。

$D_4 D_3 = \quad \times 0 \quad 01 \quad 11$

校验位选择: 无校验 奇校验 偶校验

③ D_2 选择停止位长度。

$D_2 = 0 \quad 1(D_1 D_0 = 00) \quad 1(D_1 D_0 \neq 00)$

停止位 = 1 位 1.5 位 2 位

④ $D_1 D_0$ 选择数据位长度。

$D_1 D_0 = 00 \quad 01 \quad 10 \quad 11$

数据位 = 5 位 6 位 7 位 8 位

(2) INT 14H 0 号子功能的执行流程:

① 截取 AL 的 $D_7 \sim D_5$ 位查表,取出相应的波特率除数→除数寄存器。

② 截取 AL 的 $D_4 \sim D_0$ 位→通信线控制寄存器。

③ 0→中断允许寄存器。

④ 取通信线状态寄存器的内容→AH。

⑤ 取调制解调器状态寄存器的内容→AL。

⑥ 执行 IRET 返回。

(3) 调用注意事项

调用 INT 14H 的 0 号子功能初始化串行口,通信波特率只有 8 种选择,奇偶校验也只有 3 种选择。从执行流程可以看出,使用 0 号功能初始化之后,8250 的内部中断将被禁止。只能采用查询方式发送和接收数据。如果在 0 号子功能初始化之后,再对中断允许寄存器和调制解调器控制寄存器写入相应的命令字,仍然能使其工作在中断方式。

2. 发送一帧数据

【INT 14H 1 号子功能】 发送一个数据。

入口参数: AH=1 发送数据。

AL=待发送的数据。

DX=0 使用主串口,DX=1 使用辅串口。

出口参数: AH 的 D_7 位为 1,表示发送失败; D_7 位为 0,表示发送成功。

INT 14H 的 1 号子功能执行流程如图 10-10 所示。从中看出:当使用 1 号子功能发

送数据时, BIOS 主动使引脚 $\overline{\text{RTS}}$ 为 0, $\overline{\text{DTR}}$ 为 0, 表示 8250 已经做好发送和接收准备, 然后测试 $\overline{\text{CTS}}$ 和 $\overline{\text{DSR}}$ 。只有当 $\overline{\text{CTS}}$ 和 $\overline{\text{DSR}}$ 都为 0, 而且发送保持寄存器空闲时, 才发送一个数据。在规定的时间内, 如果 $\overline{\text{CTS}}$ 和 $\overline{\text{DSR}}$ 不为 0, 或者发送保持寄存器不为空, BIOS 即认为通信联络不畅或者是 8250 有故障。随即使 AH 的 D₇ 位为 1, 返回, 宣告数据发送失败。

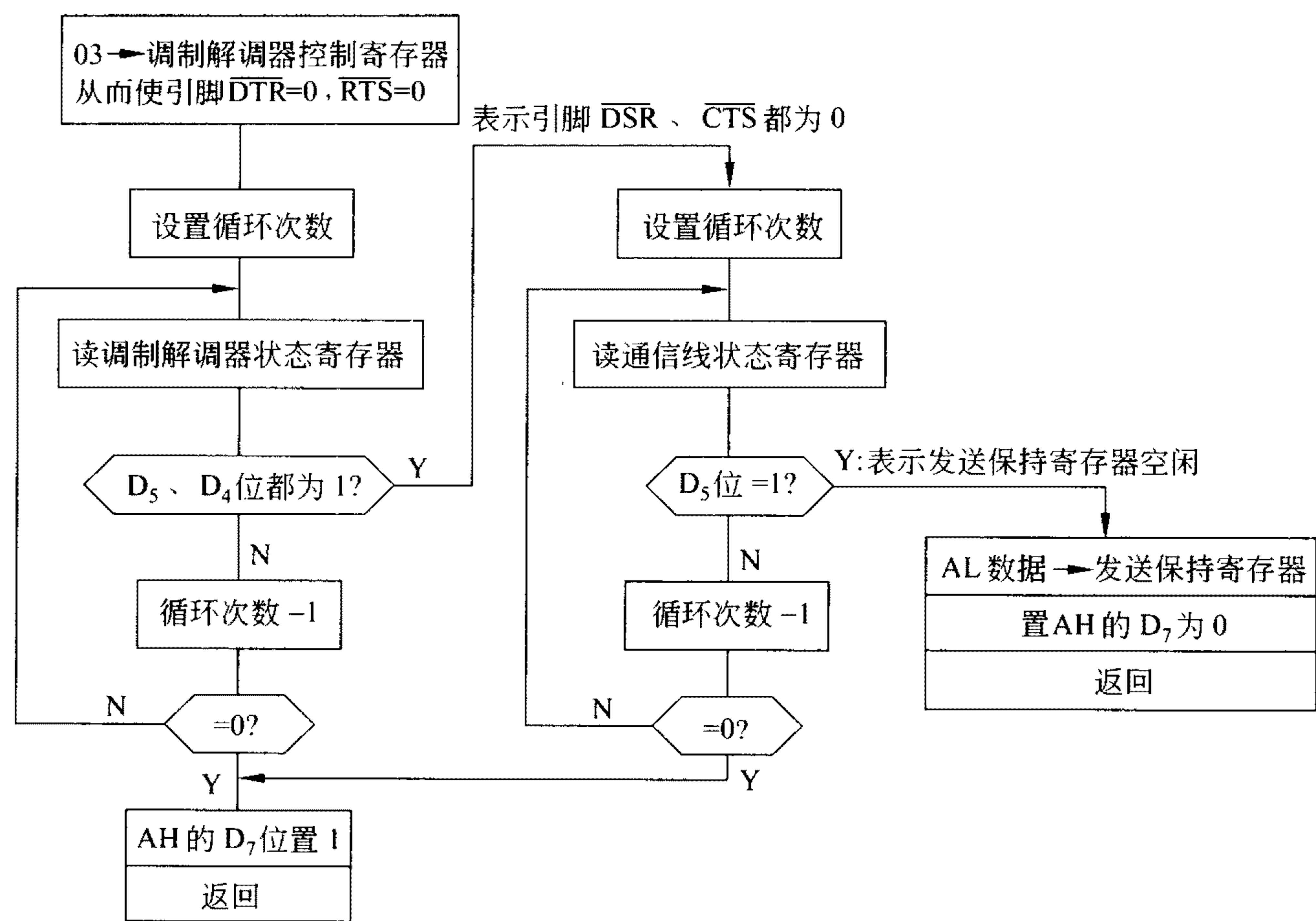


图 10-10 INT 14H 1 号子功能执行流程

3. 接收一帧数据

【INT 14H 2 号子功能】 接收一个数据。

入口参数: AH=2 接收数据。

DX=0 使用主串口, DX=1 使用辅串口。

出口参数: AH 的 D₇ 位为 1, 表示接收失败。

AH 的 D₇ 位为 0, 表示接收成功。此时 AL 中为接收的数据, AH 的 D₁~D₁ 位为接收数据的错误标志。

INT 14H 的 2 号子功能执行流程如图 10-11 所示, 从中看出: 当使用 2 号子功能接收数据时, BIOS 先使引脚 $\overline{\text{DTR}}$ 为 0, 然后测试 $\overline{\text{DSR}}$ 。只有当 $\overline{\text{DSR}}$ 为 0, 而且通信线状态寄存器表明一帧数据已经接收完毕, 此时 BIOS 才读取接收缓冲寄存器的内容送 AL。在规定的时间内, 如果 $\overline{\text{DSR}}$ 不为 0, 或者一帧数据没有收到, BIOS 即认为通信联络不畅或者是 8250 有故障。随即使 AH 的 D₇ 位为 1, 返回, 宣告数据接收失败。基于以上分析, 当使用 BIOS 通信程序发送和接收数据时, 必须具备相应的外部环境。

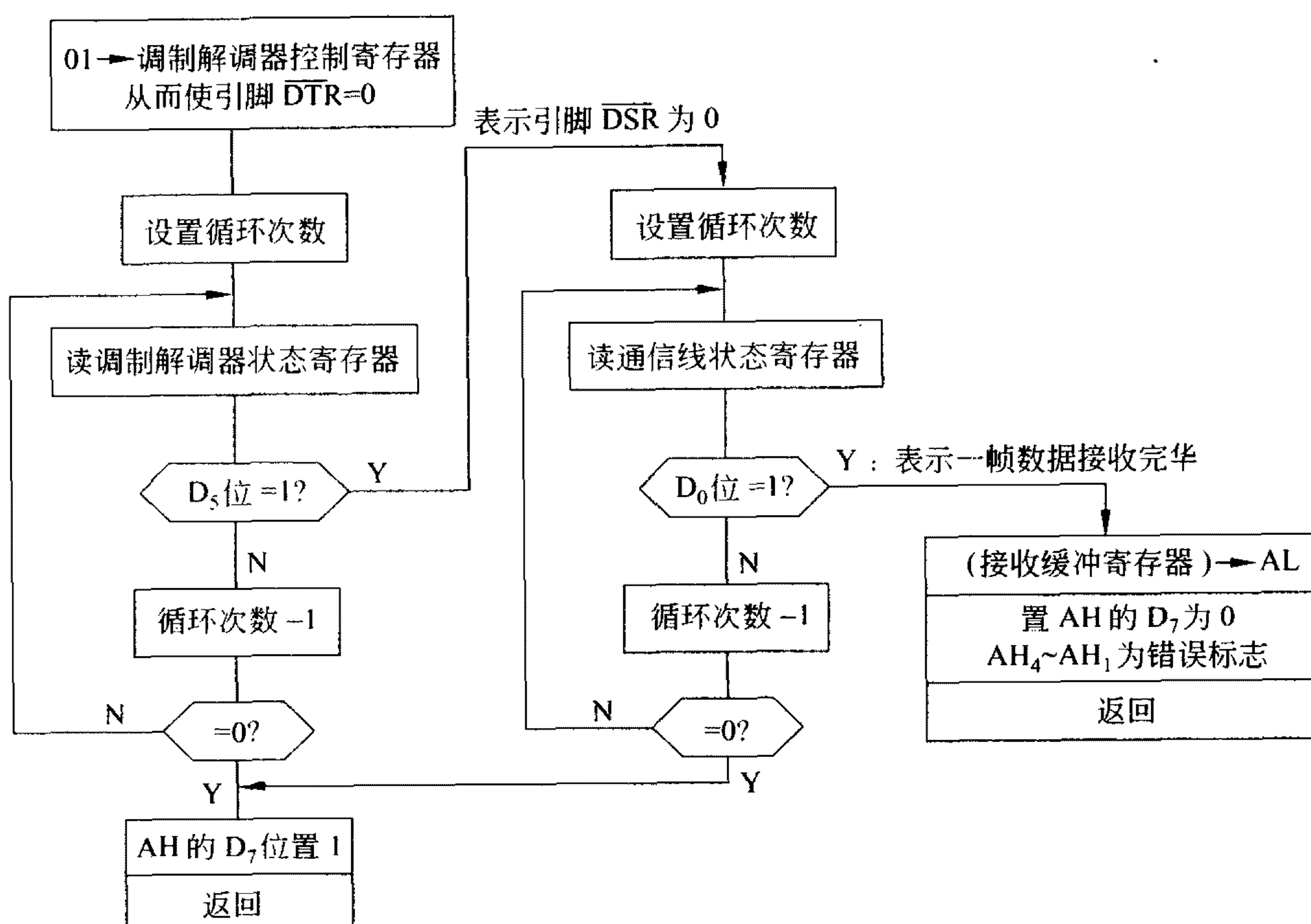


图 10-11 INT 14H 2 号子功能执行流程

4. 测试通信线状态

【INT 14H 3 号子功能】 测试通信线状态。

入口参数：AH=3 测试通信线状态。

DX=0 使用主串口, DX=1 使用辅串口。

出口参数：AH=通信线状态寄存器内容。

AL=调制解调器状态寄存器内容。

INT 14H 的 3 号子功能执行流程如下：

- ① 取通信线状态寄存器内容→AH。
- ② 取调制解调器状态寄存器内容→AL。
- ③ 返回。

10.3.2 串行通信的外部环境

外部环境就是串口连接器的外部连线方式。连线方式与串口的通信方式有关,和编程时使用的通信手段有关(直接对端口操作? 还是使用 BIOS 通信软件?)

从图 10-10 看出,使用 BIOS 通信软件发送数据时, BIOS 主动使 $\overline{\text{DTR}}=0$ 、 $\overline{\text{RTS}}=0$, 然后测试 $\overline{\text{CTS}}$ 、 $\overline{\text{DSR}}$, 当确知对方端口也做好了接收准备 ($\overline{\text{CTS}}=0$) 和发送准备 ($\overline{\text{DSR}}=0$), 并且本地端口发送保持寄存器空闲时才发送一个数据。

从图 10-11 看出,使用 BIOS 通信软件接收数据时, BIOS 主动使 $\overline{\text{DTR}}=0$, 然后测试 $\overline{\text{DSR}}$, 在确知对方端口已经做好发送准备 ($\overline{\text{DSR}}=0$), 且本地端口一帧数据已收完时, 才读取

接收缓冲寄存器的内容送 AL。综上所述, BIOS 通信软件是一个全双工的通信软件, 发送和接收之前都要使用联络线与对方端口“握手”, 只有联络畅通, 才能发送或接收数据!

对端口直接操作, 发送和接收数据时, 程序也可以仿照 BIOS 先行“握手”联络, 然后再发送和接收数据, 也可以不进行联络, 直接发送、接收。

图 10-12 以 25 芯连接器为例, 画出了几种接线方式, 其中点-点全双工、单工通信, 按照“有联络线”的方式接线。如果对端口直接操作发送和接收数据, 而且程序中也不查询联络线, 可以按“无联络线”方式接线。

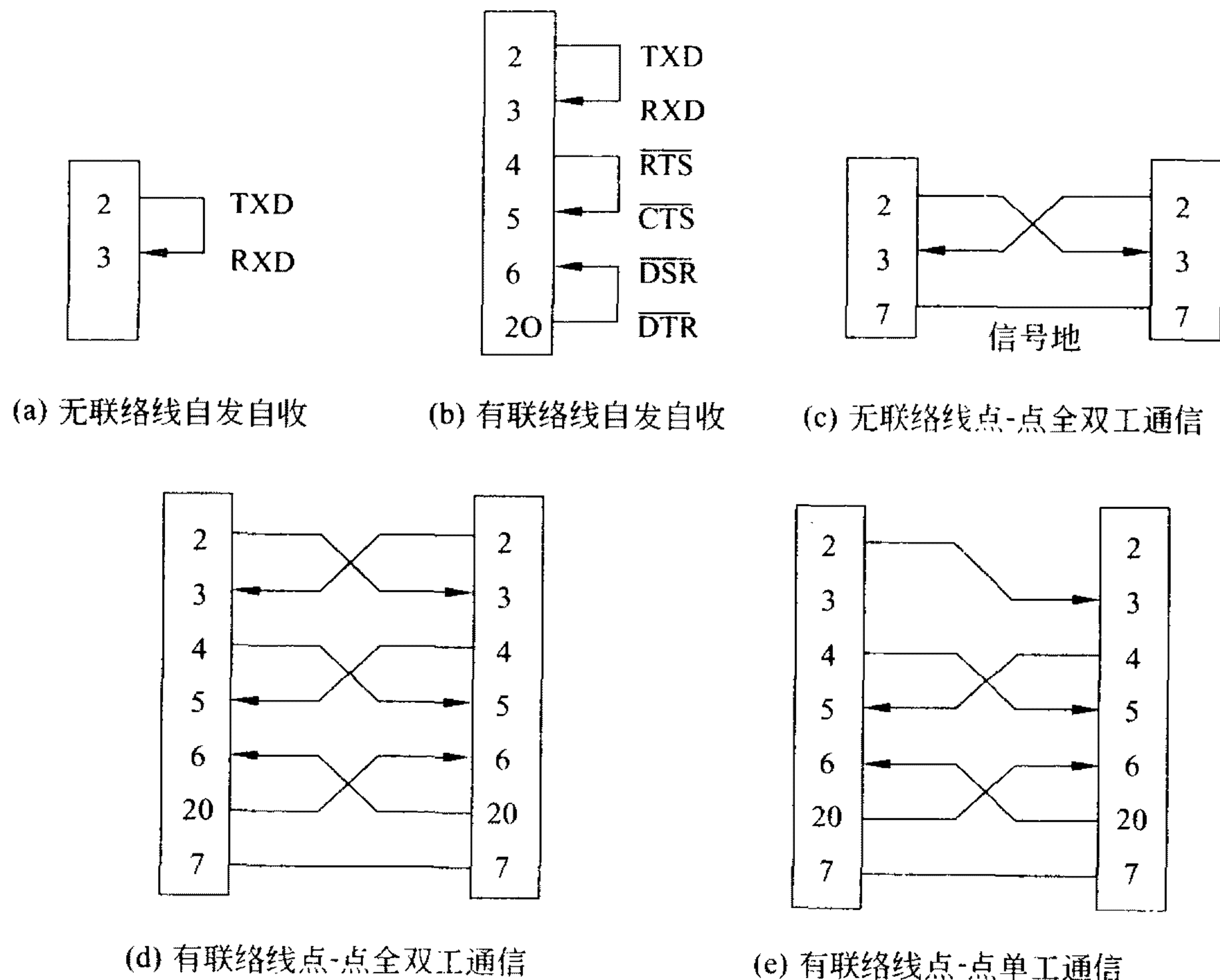


图 10-12 串行通信 RS232 连接器接线方式

高级语言例如 Turbo C 语言中的 bioscom 函数就是调用 BIOS 通信软件实现通信的, 因此用 Turbo C 语言编程实现通信, 就需要有图 10-12(d) 的外部环境。

10.3.3 串行通信程序设计

利用微型计算机系统串行口设计串行通信程序的时候, 应首先根据课题要求(自发自收、单工通信还是全双工通信)和欲采用的编程手段(对端口直接操作还是调用 BIOS 通信软件), 完成 RS-232C 连接器的连接, 创建正确的串行通信外部环境。

实验证明: 当 8250 设置为内环自检方式(初始化编程使调制解调器控制寄存器 D₄ 位置 1)的时候, 8250 无法提出中断请求, 而且引脚 RTS、CTS、DTR、DSR 在芯片内部似乎是被“切断”了。有鉴于此, 当 8250 设置为内环自检方式的时候, 只能采用查询方式, 而且只能采用对端口直接操作的编程手段, 完成数据的发送和接收。

【例 10.3.1】 对主串口进行外环自动测试, 将下列测试电文 10 行, 经主串口发出, 通过外环短路线接收, 显示在屏幕上, 测试电文如下:

THE QUICK BROWN FOX JUMPS OVER LAZY DOG

【设计思路】

(1) 电文译为：狡猾的褐色狐狸越过懒狗的背。它是国际电报通信中，常用的测试电文，电文包含了英语 26 个字母，又称狐狸电文。在一条报路上，长时间的循环发送这条电文，接收方统计在一定时间内的差错率即可知道该报路的通信质量。

(2) 电文必须逐个字符发送，为了简化程序设计，发送字符和接收字符均采用查询方式，发送前，先读取通信线状态寄存器，查询发送保持寄存器是否为空，接收前先读取通信线状态寄存器，查询一帧数据是否接收完毕。

(3) 本例采用两种方法编程：

1031_1. ASM, 直接访问 8250 端口寄存器，程序运行前 RS232 连接器按图 10-12(a) 接线，没有使用联络线。

1031_2. ASM, 调用 BIOS 通信软件，程序运行前需按图 10-12(b) 接线，准备好自发自收的外部环境。

程序框图如图 10-13 所示。

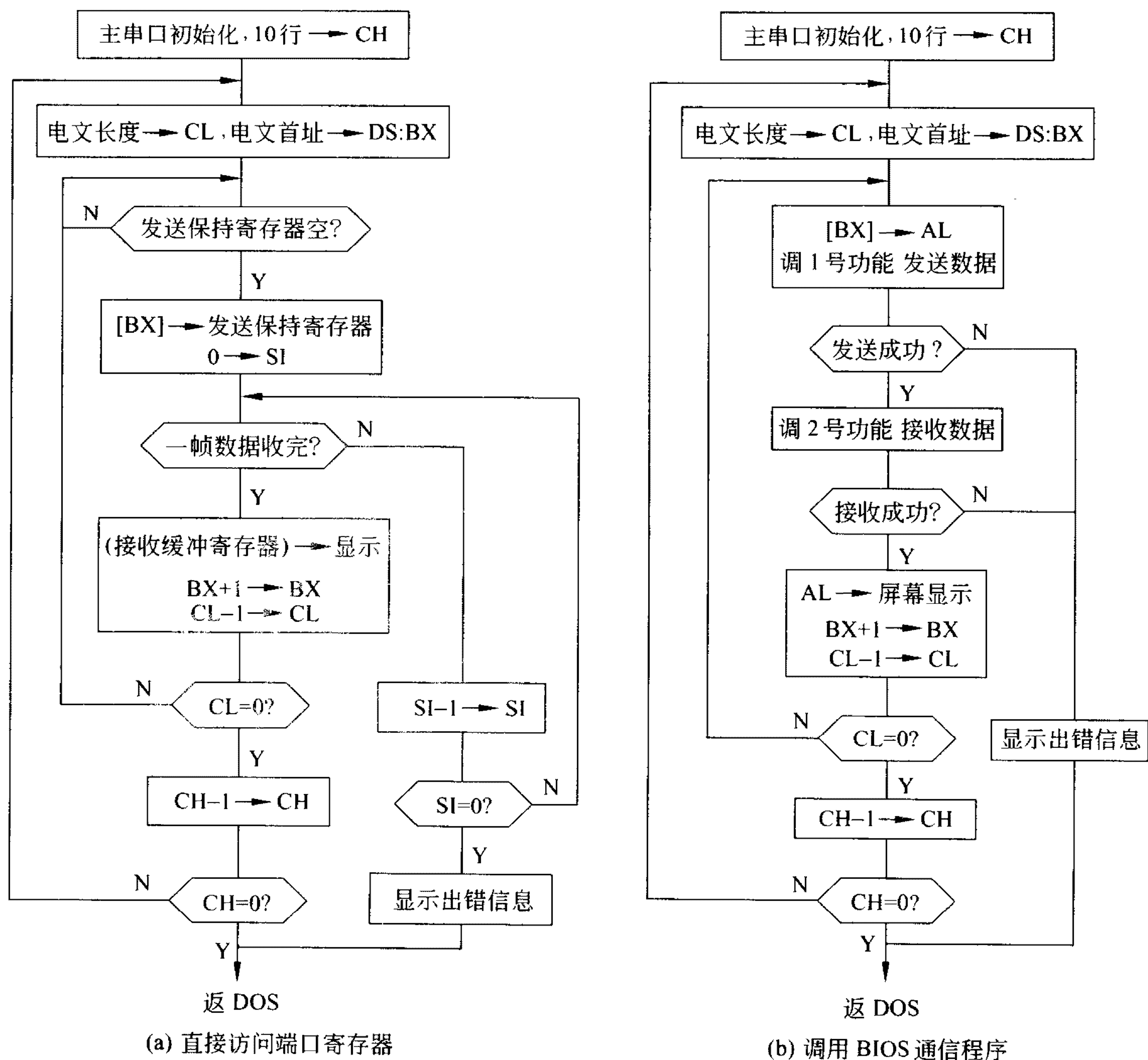


图 10-13 【例 10.3.1】程序框图

【1031_1·ASM 程序清单】

```

;FILENAME: 1031_1.ASM
DATA SEGMENT
TEXT DB 'THE QUICK BROWN FOX JUMPS OVER LAZY DOG'
      DB 0DH,0AH
LLL EQU $-TEXT
ERROR DB 'COM1 BAD !',0DH,0AH,'$'
DATA ENDS
CODE SEGMENT
      ASSUME CS: CODE,DS: DATA
BEG:  MOV AX,DATA
      MOV DS,AX
      CALL I8250 ;主串口初始化
      MOV CH,10 ;10行送CH
AGAIN: MOV CL,LLL ;电文长度送CL
      MOV BX,OFFSET TEXT
TSCAN: MOV DX,3FDH
      IN AL,DX
      TEST AL,20H ;发送保持寄存器空?
      JZ TSCAN ;否
      MOV AL,[BX] ;取字符
SEND:  MOV DX,3F8H
      OUT DX,AL ;送主串口数据寄存器
      MOV SI,0
RSCAN: MOV DX,3FDH
      IN AL,DX
      TEST AL,01H ;一帧数据收完否?
      JNZ REVEICE ;收完转移
      DEC SI
      JNZ RSCAN
      JMP DISPERR ;超时,转出错处理
REVEICE: MOV DX,3F8H
      IN AL,DX ;读数据寄存器
      AND AL,7FH
DISP:  MOV AH,2
      MOV DL,AL
      INT 21H ;屏幕显示
      INC BX
      DEC CL ;计数
      JNZ TSCAN
      DEC CH ;行计数
      JNZ AGAIN
      JMP RETURN

```



```
DISPERR:  MOV      AH,9
          MOV      DX,OFFSET ERROR
          INT      21H          ;显示出错信息
RETURN:   MOV      AH,4CH
          INT      21H          ;返回 DOS
;-----
I8250     PROC          ; 主串口初始化子程序
          MOV      DX,3FBH
          MOV      AL,80H
          OUT      DX,AL        ;寻址位置 1
          MOV      DX,3F9H
          MOV      AL,00H
          OUT      DX,AL        ;写除数高 8 位
          MOV      DX,3F8H
          MOV      AL,60H
          OUT      DX,AL        ;写除数低 8 位
          MOV      DX,3FBH
          MOV      AL,03H
          OUT      DX,AL        ;无校验传送,8 位数据
          MOV      DX,3F9H
          MOV      AL,00H
          OUT      DX,AL        ;禁止 8250 内部中断
          MOV      DX,3FCH
          MOV      AL,0
          OUT      DX,AL        ;8250 收发方式,禁止中断
          RET
I8250     ENDP
CODE      ENDS
          END      BEG
```

【1031_2. ASM 程序清单】

```
          ;FILENAME: 1031_2. ASM
DATA      SEGMENT
TEXT      DB          'THE QUICK BROWN FOX JUMPS OVER LAZY DOG'
          DB          0DH,0AH
LLL       EQU          $-TEXT
ERROR     DB          'COM1 BAD !',0DH,0AH,' $ '
DATA      ENDS
CODE      SEGMENT
          ASSUME      CS: CODE,DS: DATA
BEG:      MOV      AX,DATA
          MOV      DS,AX
          CALL      I8250        ;主串口初始化
          MOV      CH,10        ;10 行送 CH
```

```

AGAIN:    MOV     CL,LLL                ;电文长度送 CL
          MOV     BX,OFFSET TEXT
SEND:     MOV     AL,[BX]              ;取数据
          MOV     AH,1
          MOV     DX,0
          INT     14H                  ;发送数据
          TEST    AH,80H               ;发送成功否?
          JNZ     DISPERR              ;失败,转出错处理
RECEIVE:  MOV     AH,2
          MOV     DX,0
          INT     14H                  ;接收一个数据
          TEST    AH,80H               ;接收成功否?
          JNZ     DISPERR              ;失败,转出错处理
          AND     AL,7FH
DISP:     MOV     AH,2
          MOV     DL,AL
          INT     21H                  ;送屏幕显示
          INC     BX
          DEC     CL                    ;计数
          JNZ     SEND
          DEC     CH                    ;行计数
          JNZ     AGAIN
          JMP     RETURN
DISPERR:  MOV     AH,9
          MOV     DX,OFFSET ERROR
          INT     21H                  ;显示出错信息
RETURN:   MOV     AH,4CH
          INT     21H                  ;返回 DOS
;-----
I8250     PROC
          MOV     AX,0083H             ;波特率 1200
          MOV     DX,0                 ;无校验,8 位数据
          INT     14H
          RET
I8250     ENDP
CODE      ENDS
          END      BEG

```

10.4 可编程串行通信接口芯片 8251A

8251A 是为 Intel 公司微处理器进行数据通信而设计的通用串行接口芯片,8251A 是可编程的同步 / 异步、接收器 / 发送器。8251A 进行同步通信时最高通信速率为 64×10^3 波特,进行异步通信时最高通信速率为 19.2×10^3 波特。8251A 没有内置的波特率

发生器, 依赖外部输入发送器时钟($\overline{\text{TxC}}$)和接收器时钟($\overline{\text{RxC}}$)。

1. 8251A 的内部结构

图 10-14 为 8251A 的内部结构。它包括 4 个主要部分: 总线接口部分由数据总线缓冲器和读/写控制两个功能块组成; 数据发送部分由发送缓冲器和发送控制两个功能块组成; 数据接收部分由接收缓冲器和接收控制两个功能块组成; 还有调制解调控制模块。

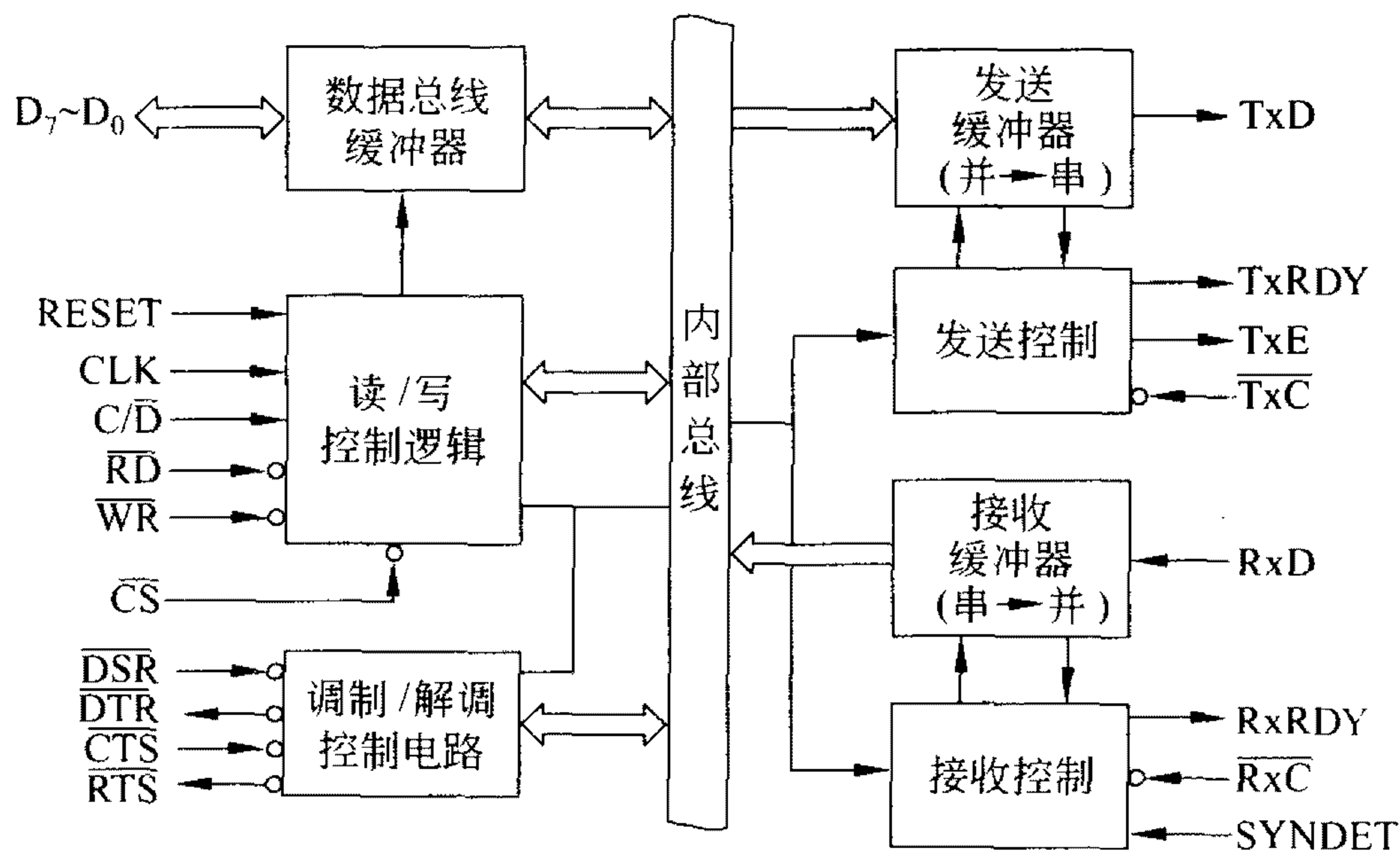


图 10-14 8251A 内部结构图

(1) 总线接口

数据总线缓冲器是 8 位的双向三态缓冲器, 它是 8251A 与 CPU 数据总线的接口模块, CPU 通过它向 8251A 写入控制命令、读取状态信息、发送和接收数据。读/写控制模块接收地址选中信号、读/写命令和控制信号。

(2) 发送器

发送缓冲器通过数据总线缓冲器接收来自 CPU 的并行数据, 在发送控制模块的作用下, 将并行数据转换成串行数据然后从 TXD 端子发出。

(3) 接收器

在接收控制模块的作用下, 接收缓冲器从 RXD 端接收串行数据并将其转换成并行数据。

(4) 调制解调控制器

该模块产生并接收调制解调器控制信号。

2. 8251A 的引脚功能

8251A 为 28 脚双列直插式封装, 使用 +5V 电源, 引脚如图 10-15 所示。

(1) 面向 CPU 一侧的引脚信号

$D_7 \sim D_0$: 双向数据线。

CLK: 时钟信号输入端, CLK 信号用来产生 8251A 内部时序。8251A 要求输入的

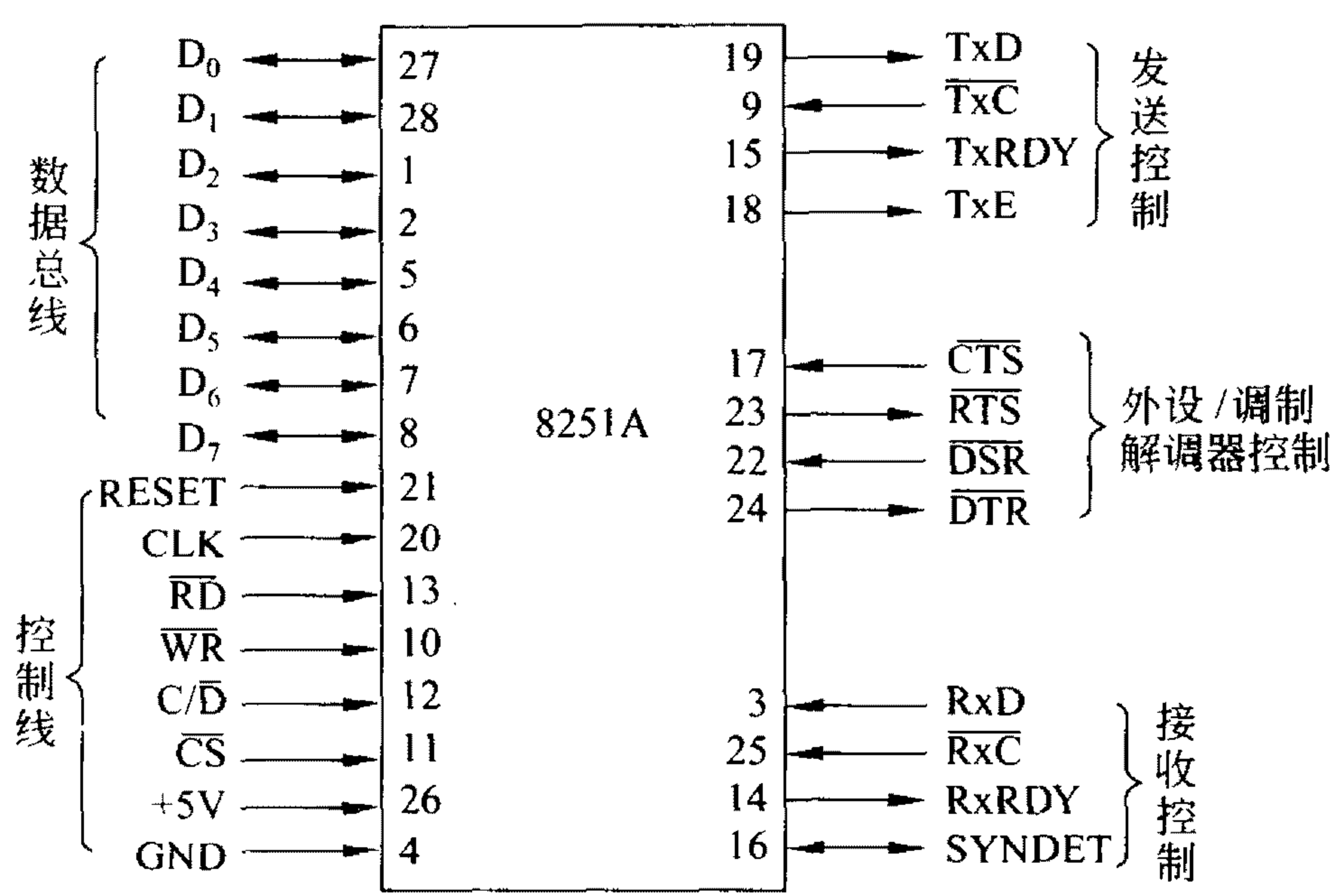


图 10-15 8251A 引脚图

时钟信号频率至少要比发送 / 接收的通信速率高出 30 倍。

RESET: 复位信号输入端,高电平使 8251A 处于空闲状态,直到写入新的控制字定义它的功能。

CS: 片选信号输入端,CS为低电平时,芯片被“选中”,加到RD、WR引脚上的信号才能起作用。

RD: 读命令输入端,低电平有效。

WR: 写命令输入端,低电平有效。

C/D: 控制 / 数据选择端,读出和写入 8251A 的信息分两类,一类是控制信息,它包括写入的命令字和读出的状态字;另一类是数据信息,即发送和接收的数据。在CS=0 的条件下,C/D=1 选中 8251A 内部的控制寄存器和状态寄存器。在CS=0 的条件下,C/D=0 选中 8251A 内部的数据寄存器。在CS=0 的条件下,C/D、RD和WR 3 个控制信号按表 10-5 所示,共同完成对 8251A 的读 / 写操作。

表 10-5 8251A 端口读/写操作

C/D	RD	WR	CS	操 作
0	0	1	0	从 8251A 读取数据
0	1	0	0	向 8251A 输出数据
1	0	1	0	从 8251A 读取状态字
1	1	0	0	向 8251A 写入控制字
×	1	1	0	无操作
×	×	×	1	禁止

(2) 发送器引脚信号

TXD: 串行数据发送端。

TXRDY(Transmitter Ready): 发送器准备好,高电平有效。该信号是 8251A 的发送中断请求信号。当初始化编程写入的工作命令字 D₀ 位(发送允许位)为 1,输入引脚

$\overline{\text{CTS}}$ 为 0, 发送缓冲器空闲, 这 3 个条件都满足的时候, TXRDY 引脚输出高电平。

TXE(Transmitter Empty): 发送器空闲, 高电平有效。发送器中并→串已完成, 一帧串行数据已发送时, 发送器空闲, TXE 引脚输出高电平。该信号也可以做为 8251A 的发送中断请求信号。和 8250 相比, TXRDY 输出高电平相当于 8250 的发送保持寄存器空闲, TXE 输出高电平相当于 8250 的发送移位寄存器空闲。

TXC: 发送器时钟信号输入端, 工作在异步方式的时候:

$$f_{\text{TXC}} = \text{波特率系数} \times \text{波特率}$$

其中, 波特率系数为 1、16、64, 由方式选择命令字确定。

(3) 接收器引脚信号

RXD: 串行数据接收端。

RXRDY(Receiver Ready): 接收器准备好, 高电平有效。该信号为 8251A 的接收中断请求信号, 当初始化编程写入的工作命令字 D_2 位为 1(允许接收)的时候, 8251A 收到一帧数据之后, RXRDY 引脚输出高电平。

RXC: 接收器时钟信号输入端, 工作在异步方式的时候:

$$f_{\text{RXC}} = \text{波特率系数} \times \text{波特率}$$

其中, 波特率系数为 1、16、64, 由方式选择命令字确定。

在大多数通信系统中, 数据发送和数据接收时的通信速率是一致的, 此时应当把 RXC 和 TXC 端子并联, 由一个波特率发生器提供发送器 / 接收器时钟信号。

SYNDET: 该引脚仅用于同步方式, 8251A 工作在内同步方式时, 该引脚为输出端, 当 8251A 检测到同步字符的时候, SYNDET 输出高电平, 表示接收已经和发送同步。8251A 工作在外同步方式时, 该引脚为输入端, 由外部输入一个正跃变作为同步信号, 在此之后 8251A 开始接收数据。

(4) 调制解调器控制信号

$\overline{\text{RTS}}$: 请求发送, 输出低电平有效。

$\overline{\text{CTS}}$: 允许发送, 输入低电平有效。

$\overline{\text{DTR}}$: 数据终端准备好, 输出低电平有效。

$\overline{\text{DSR}}$: 数据设备准备好, 输入低电平有效。

初始化编程时, 使工作命令字的 D_5 位置 1, 就能使 $\overline{\text{RTS}}$ 端子输出低电平, D_1 位置 1, 就能使 $\overline{\text{DTR}}$ 端子输出低电平, 从而向外部通报 8251A 做好了数据发送和接收的准备工作。

$\overline{\text{CTS}}$ 、 $\overline{\text{DSR}}$ 这两个端子用来接收外部信号, 外部向 $\overline{\text{CTS}}$ 输入低电平是 8251A 数据发送的必要条件之一, 程序员读取 8251A 的状态信息可以查询 $\overline{\text{DSR}}$ 端子上的输入电平。

8251A 与系统总线的连接如图 10-16 所示。8251A 工作时需要由外部时钟源提供时钟信号, 由外部的波特率发生器提供发送器 / 接收器时钟

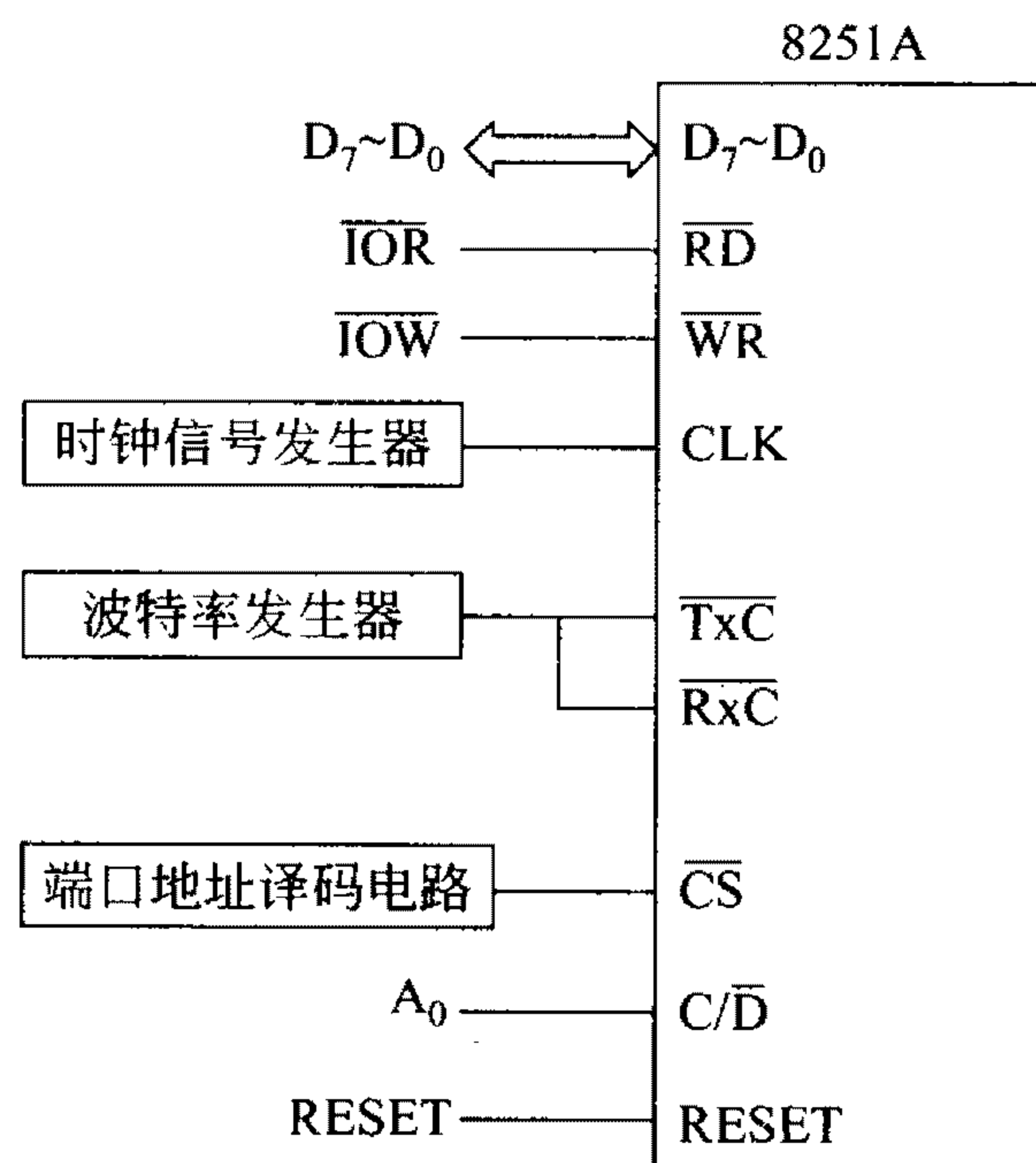


图 10-16 8251A 与系统总线连接图

信号。8251A 有两个端口,使 $\overline{CS}=0, C/\overline{D}=1$ 的端口地址访问控制端口/状态端口,使 $\overline{CS}=0, C/\overline{D}=0$ 的端口地址,访问数据端口。如果 C/\overline{D} 接到地址线 A_0 ,则 8251A 的控制端口/状态端口为奇地址,数据端口为偶地址。

3. 8251A 工作原理

8251A 能够进行同步方式的数据发送与接收,也能进行异步方式的数据发送与接收,下面简要叙述其工作过程。

(1) 同步方式数据发送

在 8251A 设定为同步方式,编程“允许发送”的时候,如果引脚 $\overline{CTS}=0$ 就启动同步发送。发送器首先发送 1 个或 2 个同步字符,然后发送数据块,并根据编程要求,为每个数据添加奇 / 偶校验位。在数据发送过程中,如果 CPU 不能及时提供待发送的数据,8251A 将自动发送同步字符。

(2) 同步方式数据接收

8251A 设定为同步方式,编程为“允许接收”的时候,8251A 首先进入搜索方式,从 RXD 端移位接收串行数据,在内同步方式下,一旦搜索到同步字符就令引脚 SYNDY 输出高电平,表示接收器已经和发送器同步,随后 8251A 接收数据,每收到一个数据就送到接收缓冲器暂存,然后使 RXRDY 引脚为高电平,状态寄存器的 RXRDY 位置 1,前者向 CPU 提出接收中断请求,后者供 CPU 查询。

(3) 异步方式数据发送

8251A 设定为异步方式,编程为“允许发送”,当引脚 $\overline{CTS}=0$ 时,就启动异步发送方式,发送缓冲器为每一个待发送的数据添加起始位,并根据编程的设定,添加奇 / 偶校验位和规定长度的停止位,一位一位地发送,当 CPU 不能及时提供待发送数据的时候,发送器自动输出停止位信号(逻辑 1)。

(4) 异步方式数据接收

8251A 设定为异步方式,编程为“允许接收”的时候,接收缓冲器就不断检测 RXD 端,一旦检测到高电平到低电平的跃变,就启动内部计数器计数,当计数到二分之一数据位时间的时候,再次采样 RXD 端,若 RXD 端仍然为低电平就表明这是一帧数据的起始位,随后以位时间间隔采样并移位接收一帧数据,去掉起始位停止位,转换成并行数据送接收缓冲器暂存,使引脚 RXRDY 输出高电平、状态寄存器 RXRDY 位置 1。前者用来向 CPU 提出接收中断请求,后者供 CPU 查询。

4. 8251A 的命令字与初始化编程

8251A 有两个命令字(方式选择命令字、工作命令字)一个状态字。两个命令字没有特征位,必须按先后顺序写入控制端口。状态端口与控制端口使用一个端口地址,从状态端口读出的是状态字。

(1) 方式选择命令字

图 10-17 为方式选择命令字的格式, $D_1 D_0 = 00$,8251A 工作在同步方式。 $D_1 D_0 \neq 00$,

8251A 工作在异步方式。工作在同步方式的时候, D_7D_6 位的 4 种组合用来定义内同步还是外同步, 以及发送同步字符的个数。工作在异步方式的时候, D_7D_6 位的组合用来选择一帧数据停止位的长度, D_1D_0 位选择波特率系数。

$$\text{波特率} = f_{\text{TXC}, \text{RXC}} \div \text{波特率系数}$$

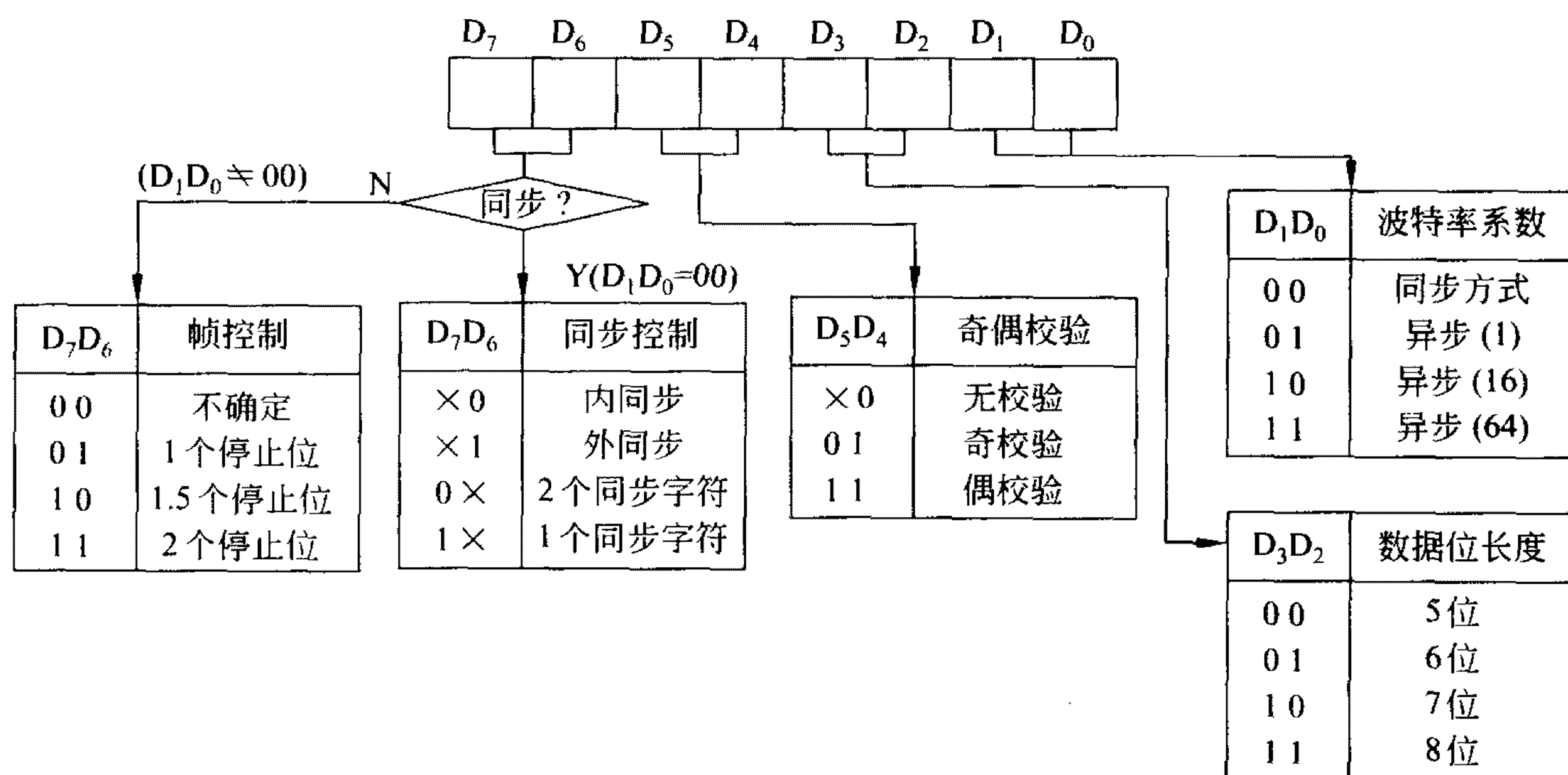


图 10-17 8251A 方式选择命令字格式

(2) 工作命令字

工作命令字用来控制 8251A 的实际操作, 其命令字格式如图 10-18 所示。

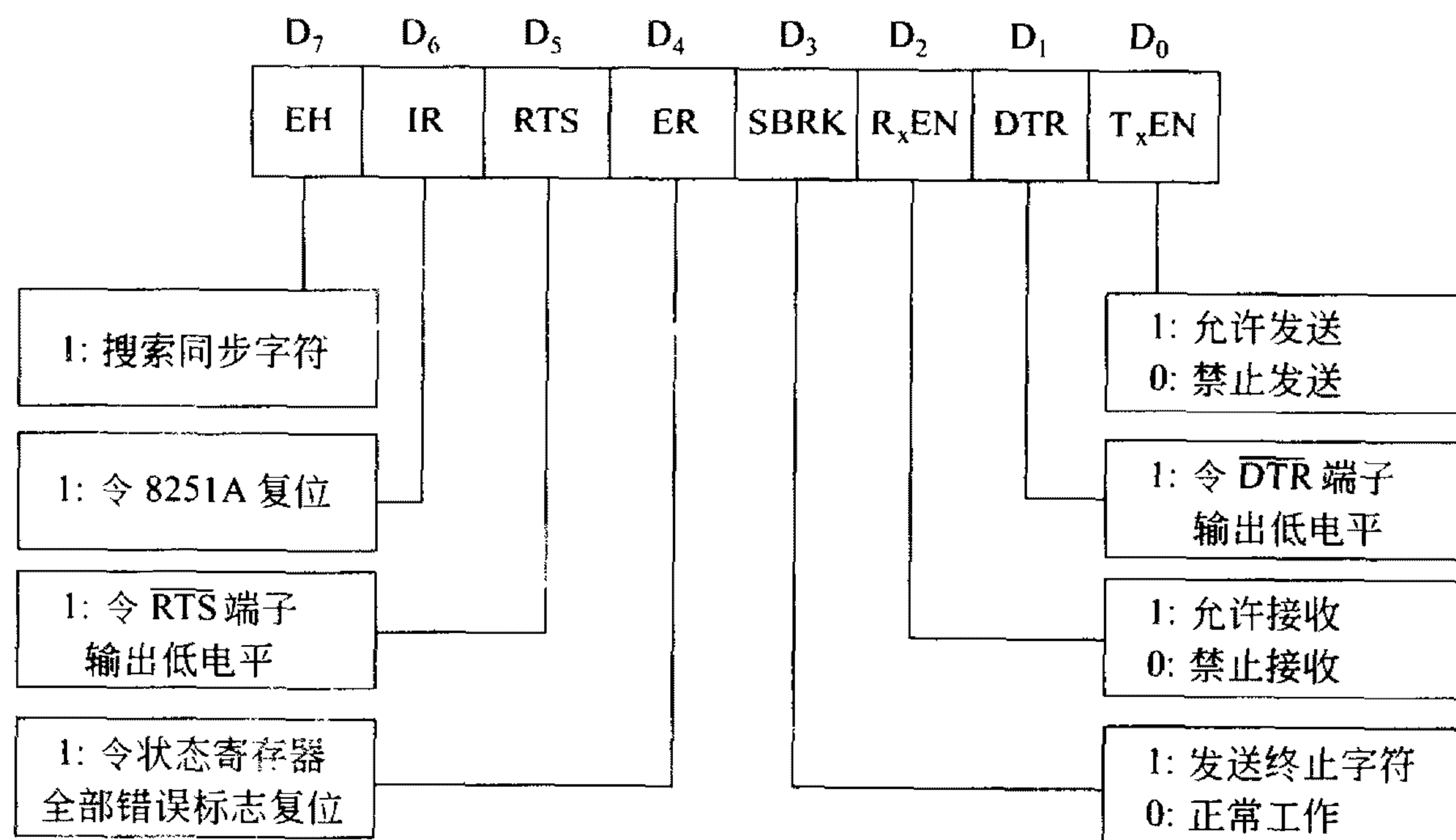


图 10-18 8251A 工作命令字格式

这里要着重说明几个问题:

① 当写入 D_6 位为 1 的命令字 (40H) 的时候, 8251A 内部复位, 进入接收“方式选择命令字”的状态。

② 当两片 8251A 进行点-点双机通信的时候, 一端的 $\overline{\text{RTS}}$ 、 $\overline{\text{DTR}}$ 端子和另一端的

CTS、DSR端子相连,此时要设置命令字的 D₅ 位、D₁ 位为 1,使RTS、DTR端子输出低电平,向对方通报本端已经做好了数据发送和接收的准备工作。

③ D₃ 位置 1,使 TXD 端输出长时间的逻辑 0,正常工作时 D₃ 位应当置 0。

④ 欲使 8251A 发送数据,D₀ 位应置 1,欲使 8251A 接收数据,D₂ 位应置 1,工作在全双工方式的时候,D₂ 位、D₀ 位应同时置 1。

(3) 状态字

状态字表示 8251A 的内部状态和个别引脚的状态电平,供 CPU 查询。状态字格式如图 10-19 所示。

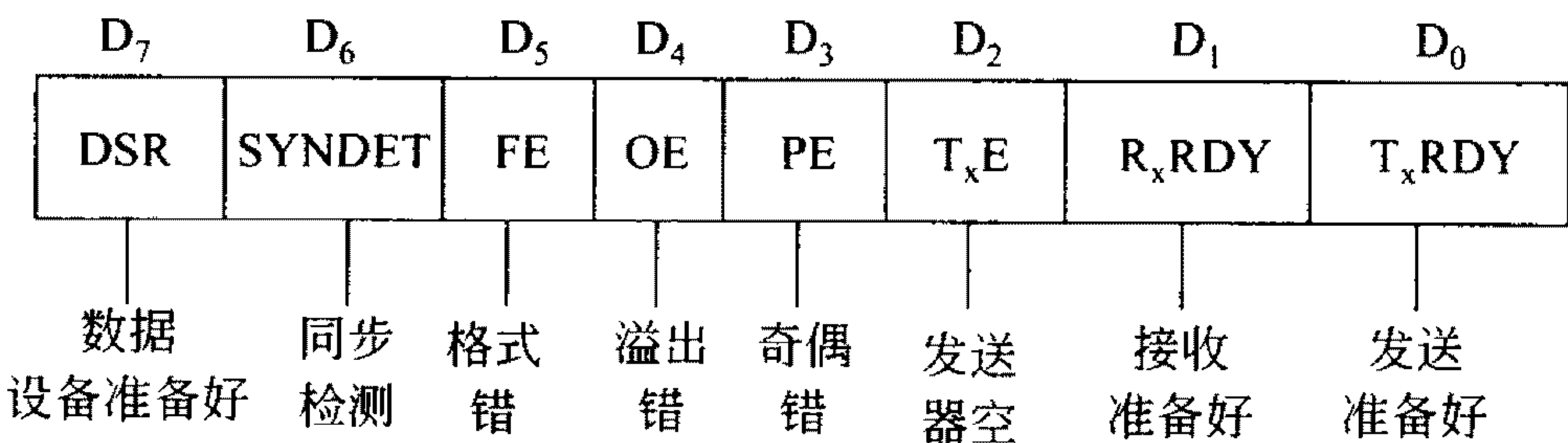


图 10-19 8251A 状态字格式

说明：

① D₇ 位为 1,表明DSR端子的输入电平为逻辑 0。

② D₆、D₂、D₁ 位与同名引脚的功能相同。

③ 在异步方式接收一帧数据的时候,没有收到规定长度的停止位,则 D₅ 位置 1,表示有帧错误(格式错)。

接收缓冲器中上一个数据没有被 CPU 读取,缓冲器又收到下一个数据,从而使上一个数据丢失,则 D₄ 位置 1,表示有溢出错。收到的一个数据中,奇偶性与编程设定的不一致则 D₃ 位置 1,表示有奇偶错。出现上述错误时,8251A 仍继续工作。

④ D₀ 位发送准备好。状态字 D₀ 位和引脚 TXRDY,虽然名称相同,但是置位的条件不一样。

引脚 TXRDY 置 1 的条件是：输入引脚CTS为 0;工作命令字 TXEN 位(D₀ 位)为 1;发送缓冲器空闲,3 个条件缺一不可。

状态字 TXRDY 位(D₀ 位)置 1 的条件是：发送缓冲器空闲。

从应用的角度讲,程序员若采用中断方式向 8251A 发送数据,首先要把引脚 TXRDY 连到 CPU 的中断请求输入端(在 PC 机中若外扩 8251A,其 TXRDY 引脚只能接到总线插槽空闲的中断请求输入端),当 CPU 响应 8251A 中断时,在服务程序中向 8251A 写入数据。若采用查询方式发送数据,可以查询状态字 TXRDY 位,只要 TXRDY 位为 1,就向 8251A 写入一个数据。

请读者不要误会,既然状态字 TXRDY 位置 1 不受引脚CTS和工作命令字 TXEN 位的制约,这是否意味着用查询方式向 8251A 发送数据的时候,引脚CTS可以不为 0? 工作命令字 TXEN 位可以不置 1 呢? 不行。不论采用查询方式还是中断方式,要完成数据发送,引脚CTS必须为 0,工作命令字 TXEN 位必须置 1。

(4) 8251A 的初始化编程

8251A 进行数据通信之前,必须完成初始化编程,图 10-20 给出了 8251A 的初始化流程。

工作在同步方式,应依次向控制端口写入方式选择命令字、1~2 个同步字符、工作命令字。

工作在异步方式,应依次向控制端口写入方式选择命令字、工作命令字。

初始化编程果真如此简单吗? 不是!

上述初始化步骤是在芯片复位的前提下进行的。怎样使芯片复位呢? 从硬件上讲,芯片加电,或 RESET 引脚输入高电平可以使芯片复位,从编程角度讲,向控制端口写入 D_6 位为 1 的工作命令字可以使芯片复位,但在实际编程中也不是一帆风顺的。

8251A 芯片使用手册中介绍:要想保证器件在接收复位命令之前处于“命令状态”,最坏情况下,也是最可靠的步骤是先向控制端口写入 3 个 0,然后再写入复位命令 40H,使器件返回到“空闲”状态。手册中还介绍:状态的变更从实际事件影响状态时算起,有 28 个时钟周期的时延! 有鉴于此,经过实验验证,我们向读者推荐异步通信的初始化编程步骤如下:

- ① 向控制端口写入 3 个 0;
- ② 向控制端口写入 40H 令 8251A 复位;
- ③ 延时(延时时间应大于 8251A 时钟周期的 28 倍),等待内部状态转换完毕;
- ④ 依次向控制端口写入方式选择命令字和工作命令字。

【例 10.4.1】 8251A 异步通信实验。

假设 PC 机 ISA 总线外扩以下实验电路,8251A 的 TXD 与 RXD 端子连接构成自发自收的实验环境。用查询方式编程将下列测试电文 10 行经 8251A TXD 端发送,RXD 端接收,最终显示在屏幕上,测试电文如下:

THE QUICK BROWN FOX JUMPS OVER LAZY DOG

【实验电路】

实验电路如图 10-21 所示。

【设计思路】

(1) 组织 8251A 的方式选择命令字

假设一帧数据中包含 8 个数据位,1 个停止位,无校验传送,而且波特率系数取 16,则方式选择命令字为 4EH。

(2) 组织 8251A 的工作命令字

本例使用 8251A 发送和接收数据,所以应当允许发送,允许接收,即工作命令字

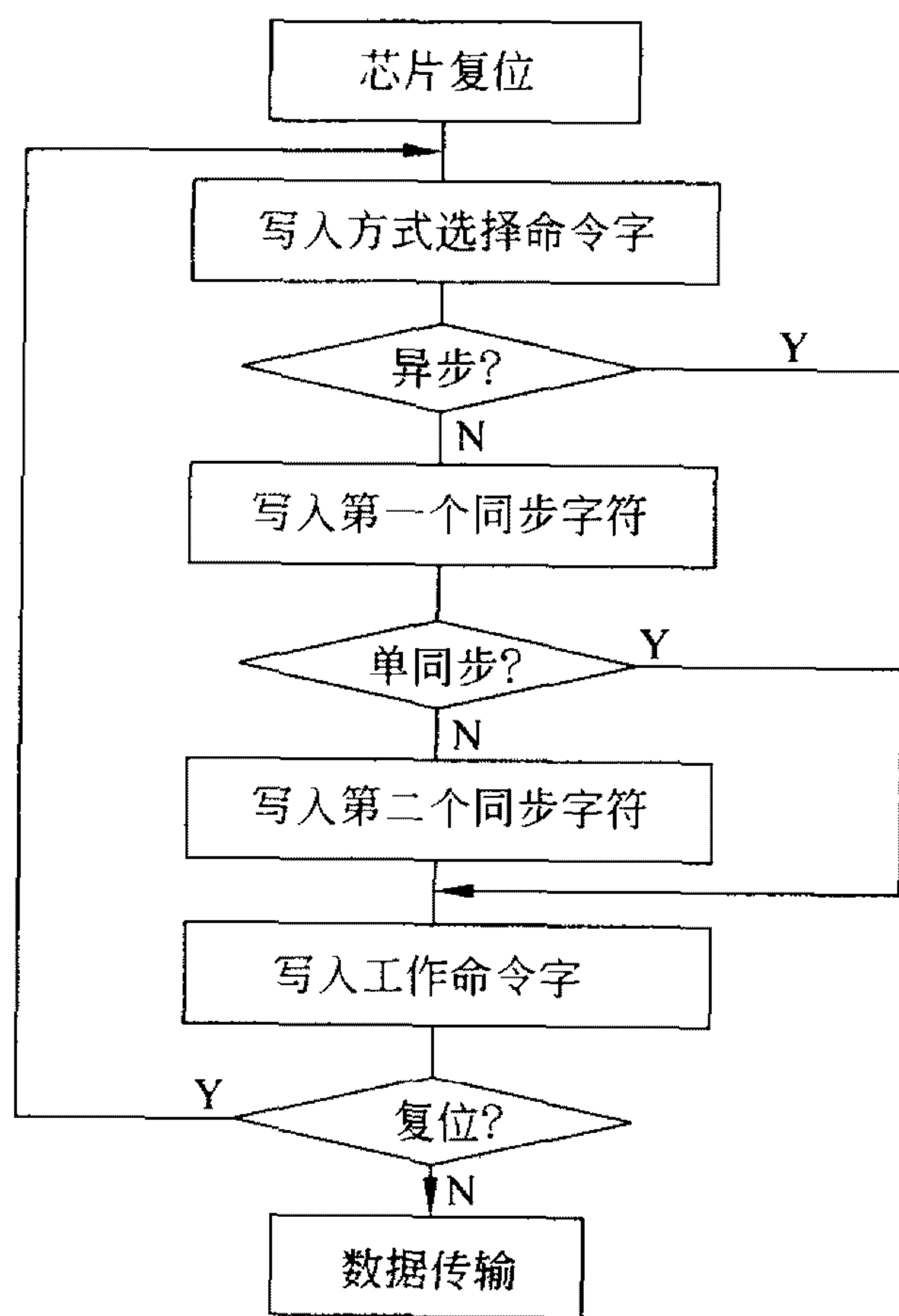


图 10-20 8251A 初始化流程

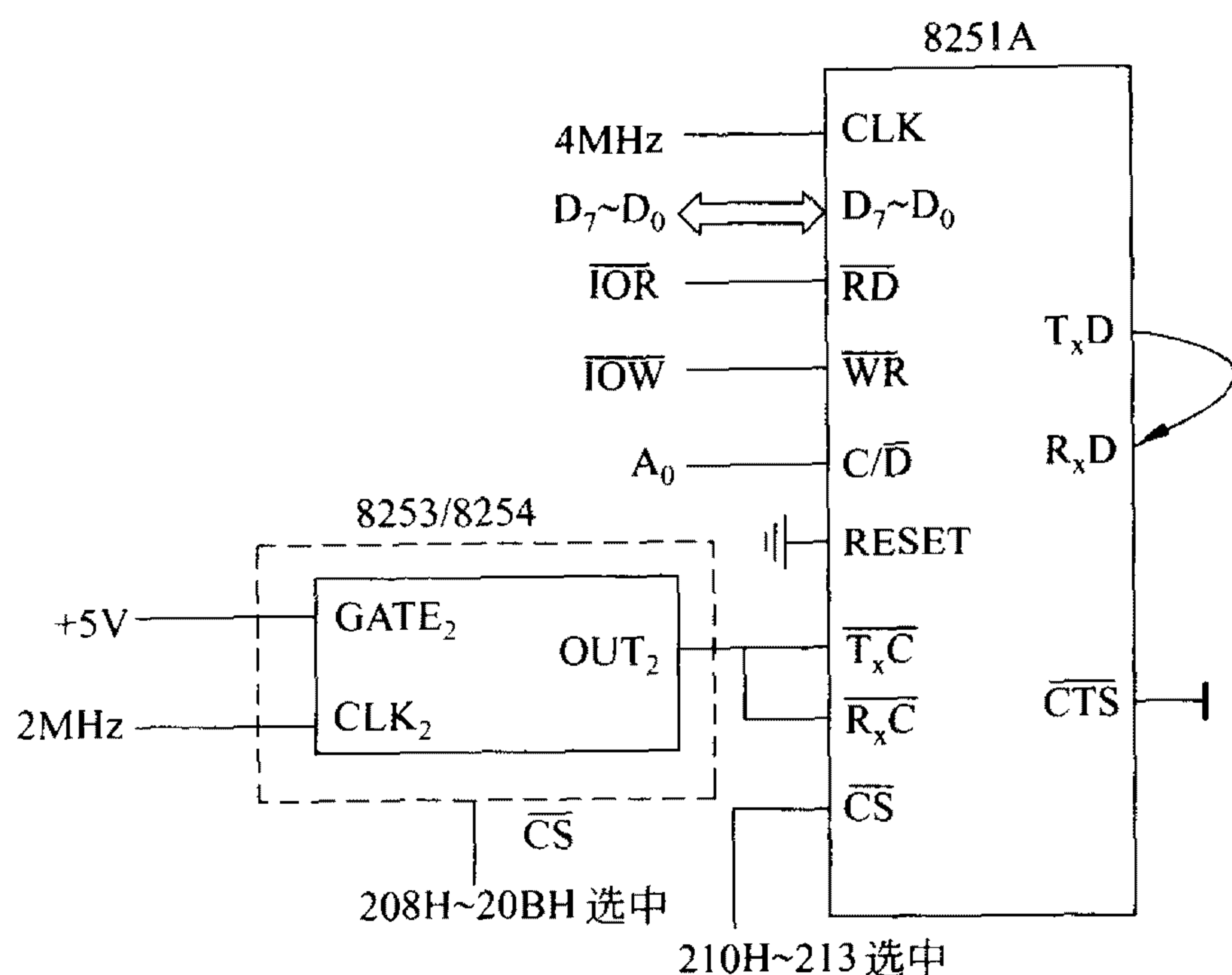


图 10-21 8251A 异步通信实验电路

TXEN 位置 1,RXE 位也应置 1,借此机会也令错误标志复位,即 ER 位置 1,故工作命令字为 15H。

(3) 计算 8254 的 2 号计数器工作参数

8251A 没有内置的波特率发生器,本例外扩一片 8254,用其 2 号计数器,产生连续方波,作为 8251A 的发送器时钟和接收器时钟信号,如果 8251A 的通信速率为 1200 波特,则发送器和接收器时钟信号的频率应为:

$$f_{OUT2} = f_{TXC,RXC} = 16 \times 1200 = 19.2\text{kHz}$$

在 $f_{CLK2} = 2\text{MHz}$ 的条件下,2 号计数器的计数初值大约为 104,初始化编程时令初值为二进制数,只写低 8 位,高 8 位由 8254 自动补 0,因此 8254 的控制字为 96H。

(4) 如前所述,为了启动 8251A 的发送器,引脚 CTS 必须为低电平,这有两种方法:其一,将 RTS(或 DTR)引脚与 CTS 引脚短路,写入工作命令字时令工作命令字 D_5 (或 D_1) 位置 1,初始化编程后,CTS 就为低电平;其二,引脚 CTS 直接接信号地,为醒目起见,我们选用第 2 种方法。

(5) 8251A 的 C/\bar{D} 引脚与地址线 A_0 相连,当 \overline{CS} 片选地址为 210H~213H 时,8251A 的控制端口地址为 211H、213H,数据端口地址为 210H、212H。

【程序清单】

```

;FILENAME: 8251A.ASM
DATA SEGMENT
MSG DB 'TEST 8251 TXD--RXD',0DH,0AH,'$'
TEXT DB 'THE QUICK BROWN FOX JUMPS OVER LAZY DOG',0DH,0AH
LENS EQU $-TEXT
ERROR DB 0DH,0AH,'8251 BAD!$'
C_8251 EQU 211H ;8251 控制端口地址
D_8251 EQU 210H ;8251 数据端口地址
C_8254 EQU 20BH ;8254 控制端口地址

```

```

D_8254      EQU      20AH                      ;8254 数据端口地址
DATA        ENDS
CODE        SEGMENT
            ASSUME    CS: CODE,DS: DATA
BEG:        MOV      AX,DATA
            MOV      DS,AX
            CALL     I8254                      ;8254 初始化
            CALL     I8251                      ;8251 初始化
            MOV      AH,9
            MOV      DX,OFFSET MSG
            INT      21H
            MOV      CH,10                      ;电文行数→CH
AGAIN:      MOV      CL,LENS                    ;一行电文的字符数→CL
            MOV      BX,OFFSET TEXT             ;电文首字符偏移地址→BX
TSCAN:      MOV      DX,C_8251
            IN       AL,DX                      ;8251 状态字→AL
            TEST     AL,01H                     ;TxRDY=1?
            JZ       TSCAN                     ;否,转移
SEND:       MOV      AL,[BX]
            MOV      DX,D_8251
            OUT      DX,AL                     ;发送数据
            MOV      SI,0
RSCAN:      MOV      DX,C_8251
            IN       AL,DX                      ;8251 状态字→AL
            TEST     AL,02H                     ;RxRDY=1?
            JNZ      REVEICE                   ;是,转移
            DEC      SI
            JNZ      RSCAN
            JMP      ERR                      ;超时,转移
REVEICE:    MOV      DX,D_8251
            IN       AL,DX                      ;接收数据
            MOV      AH,0EH
            INT      10H                      ;屏幕显示
            INC      BX
            DEC      CL
            JNZ      TSCAN
            DEC      CH                      ;行计数
            JNZ      AGAIN
            JMP      EXIT
ERR:        MOV      AH,9
            MOV      DX,OFFSET ERROR
            INT      21H
EXIT:       MOV      AH,4CH
            INT      21H
;-----
I8254      PROC                      ;8254_2 号计数器初始化

```

```

MOV      DX,C_8254
MOV      AL,96H          ;控制字
OUT      DX,AL           ;→控制寄存器
MOV      DX,D_8254
MOV      AL,104          ;计数初值→AL
OUT      DX,AL           ;写入初值低 8 位
RET
I8254    ENDP
;-----
I8251    PROC              ; 8251 初始化
MOV      CX,3
MOV      AL,0
MOV      DX,C_8251
AGA:     OUT      DX,AL    ;3 个 0 写入控制端口
LOOP     AGA
MOV      AL,40H
OUT      DX,AL           ;写入复位命令字
CALL     DELAY           ;延时
MOV      DX,C_8251
MOV      AL,01001110B
OUT      DX,AL           ;写入方式选择命令字
MOV      AL,00010101B
OUT      DX,AL           ;写入工作命令字
RET
I8251    ENDP
;-----
DELAY    PROC              ; 延时 10 微秒
MOV      AH,86H
MOV      CX,0
MOV      DX,10
INT      15H
RET
DELAY    ENDP
CODE     ENDS
END      BEG
```

说明：如果主机装有 Windows 2000 以上的操作系统,上述程序的 DELAY 子程序需要参考第 5 章例题 5. 10. 9,改用“INT 21H”的 2CH,2DH 功能调用。

习 题

- 1. 异步通信的特点是什么?
- 2. 同步通信的特点是什么?

3. 异步通信一帧字符的格式是什么?
4. 设异步通信一帧字符有八个数据位,无校验,一个停止位,如果波特率为 9600,则每秒能传输多少个字符?
5. 单工、半双工、全双工通信方式的特点是什么?
6. 在 RS-232C 接口标准中,引脚 TXD、RXD、 $\overline{\text{RTS}}$ 、 $\overline{\text{CTS}}$ 、 $\overline{\text{DSR}}$ 、 $\overline{\text{DTR}}$ 的功能是什么?
7. 分别叙述 TTL 和 RS-232C 的电平标准,通常采用什么器件完成两者之间的电平转换?
8. 8250 芯片通信线控制寄存器中的寻址位有什么作用? 在初始化编程时,应如何设置?
9. 利用系统机进行串行通信时,对串口初始化编程有哪些方法? 具体的初始化编程步骤是什么?
10. 利用 8250 查询方式发送字符时,在什么情况下可以查询发送保持寄存器空闲? 在什么情况下必须查询发送移位寄存器空闲?
11. 用系统机串行口采用中断方式完成字符发送和接收,编程时应采取哪些措施?
12. 利用系统机串行口进行短距离全双工点对点通信时,应具备什么样的外部环境? 为什么?
13. 8251A 可工作在什么工作方式?
14. 简述 8251A 内部各功能模块的作用。
15. 8251A 数据发送的条件是什么? 工作在异步通信方式,初始化编程有哪些步骤?
16. 8250 和 8251A 在通信方式和波特率设置方面有什么不同?
17. 8250 和 8251A 在发送器和接收器时钟信号的产生方式上有什么不同?

并行 I/O 接口

并行通信是同时将数据的所有位进行传输,传输速度比串行通信快。但是,因其硬件开销大,系统费用高,因而不适用于远距离数据传输。

一般并行接口应具有以下功能:

- ① 具有一个或多个 I/O 端口。
- ② 每个端口应具备与 CPU 及 I/O 设备进行联络控制的功能。
- ③ CPU 与并行 I/O 接口可用查询方式或中断方式交换信息。

11.1 可编程并行 I/O 接口芯片 8255A

8255A 是目前应用最广的可编程并行接口芯片之一,具有 40 条引脚,使用单一+5V 电源,双列直插式封装。

11.1.1 8255A 的内部结构及外部引脚

1. 8255A 内部结构

8255A 的内部结构如图 11-1 所示。

8255A 芯片内部有 3 个 8 位的输入/输出端口,即 A 端口、B 端口和 C 端口。从内部控制的角度来讲,可分为两组: A 组和 B 组。A 组控制模块管理 A 端口和 C 端口的高 4 位($PC_7 \sim PC_4$),B 组控制模块管理 B 端口和 C 端口的低 4 位($PC_3 \sim PC_0$)。

(1) 数据总线缓冲器

数据总线缓冲器是双向三态 8 位缓冲器,可以直接与系统的数据总线相连,实现 CPU 和端口之间的信息交换。

(2) 读/写控制模块

地址线 A_1, A_0 , 片选信号 \overline{CS} 和读、写控制信号 ($\overline{RD}, \overline{WR}$), 完成内部端口选择和读/写操作。

(3) A 组和 B 组控制模块

A 组控制模块: 管理 A 端口及 C 端口的高 4 位($PC_7 \sim PC_4$)。

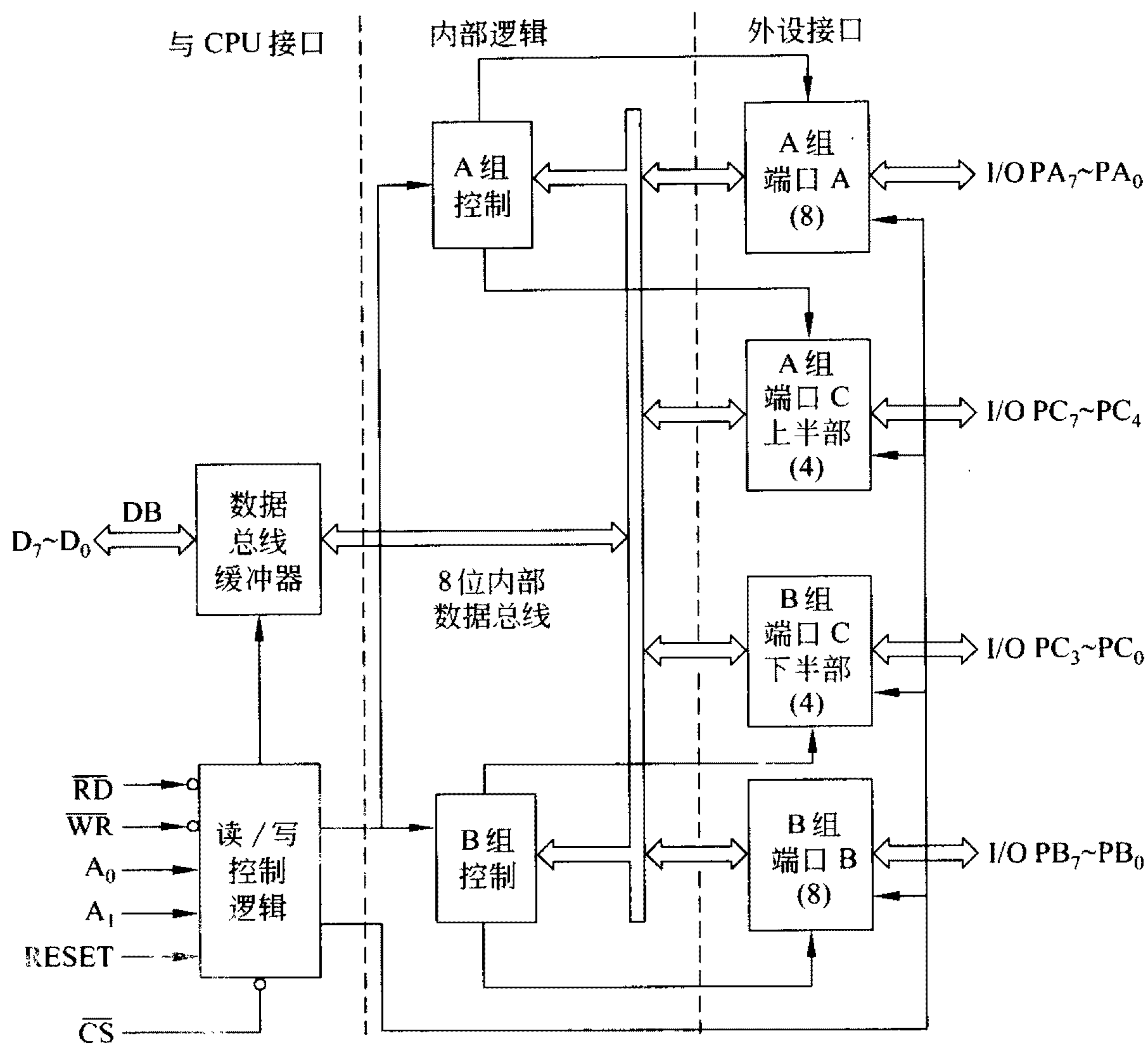


图 11-1 8255A 的内部结构框图

B 组控制模块：管理 B 端口及 C 端口的低 4 位 ($PC_3 \sim PC_0$)。

(4) I/O 端口

I/O 通道由 3 个 8 位的端口寄存器，即 A 端口、B 端口和 C 端口组成，A 端口、B 端口和 C 端口都可编程为输入/输出，而且都有数据锁存功能。C 端口可通过编程分为两个 4 位口，每一个 4 位端口都可定义为输入端口或输出端口，用于传送数据。

2. 8255A 外部引脚

8255A 的外部引脚如图 11-2 所示。

$PA_7 \sim PA_0$ ：A 端口的 I/O 数据线。

$PB_7 \sim PB_0$ ：B 端口的 I/O 数据线。

$PC_7 \sim PC_0$ ：C 端口的 I/O 数据线。

$D_7 \sim D_0$ ：双向数据线。与系统数据线相连，通过它 CPU 向 8255A 的端口写入控制字或数据；通过它 CPU 从 8255A 端口读取数据。

\overline{CS} ：片选信号。 $\overline{CS}=0$ ，选中 8255A 芯片。

A_1 、 A_0 ：端口选择信号。

\overline{RD} ：读信号，低电平有效。 \overline{RD} 有效，CPU 从 8255A 的端口读取数据。

\overline{WR} ：写信号，低电平有效。 \overline{WR} 有效，CPU 向 8255A 端口写入控制字或数据。

RESET：复位信号，高电平有效。RESET 信号有效，所有内部寄存器被清零，同时，

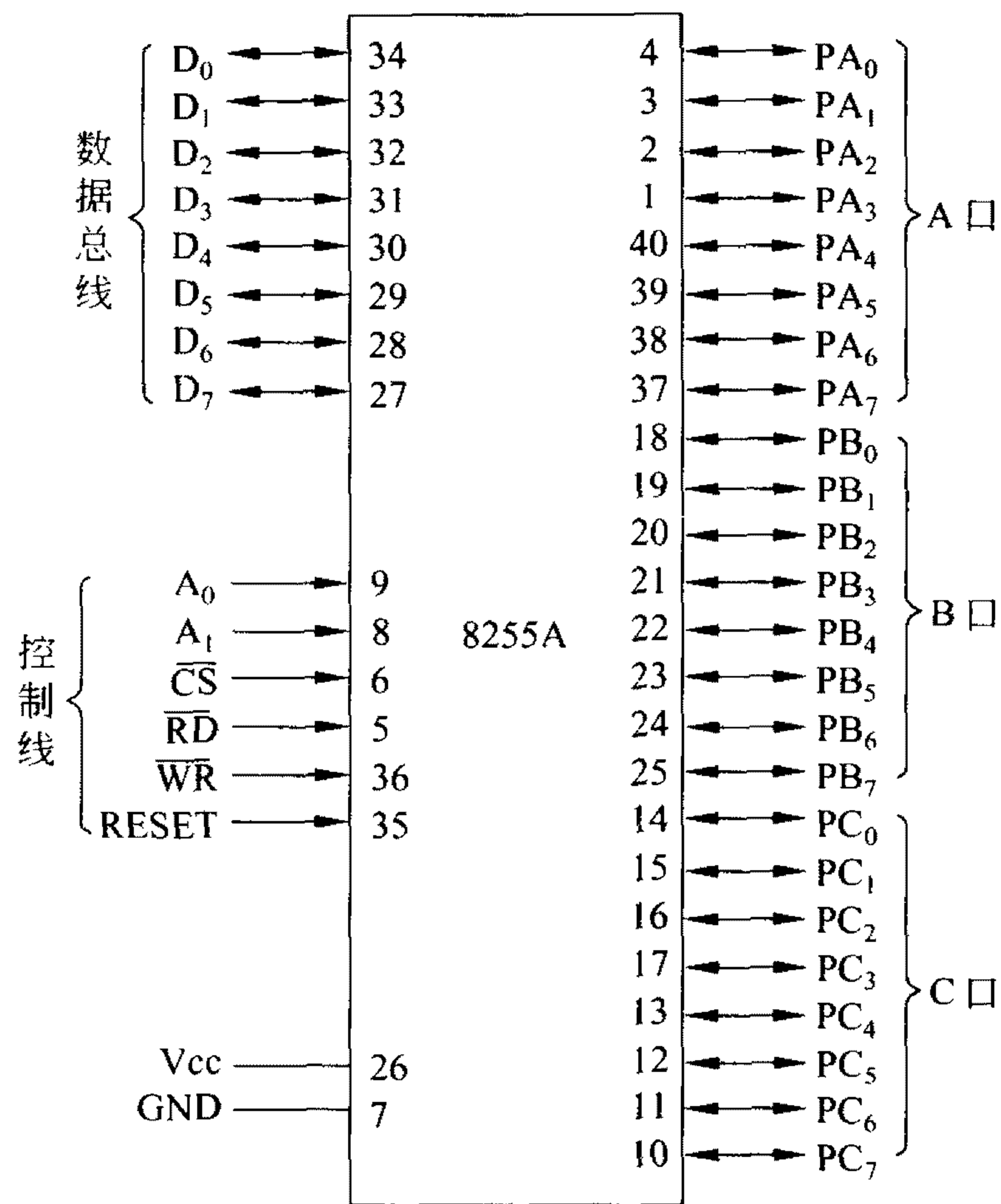


图 11-2 8255A 的外部引脚

3 个数据端口被自动设为输入端口。

3. 8255A 端口编址与读/写操作

A₁、A₀ 是端口选择信号，当 \overline{CS} 有效时，由 A₁、A₀ 的组合选择 8255A 数据寄存器和控制寄存器。8255A 的端口编址如下：

- A₁ A₀ = 00，选中 A 端口数据寄存器。
- A₁ A₀ = 01，选中 B 端口数据寄存器。
- A₁ A₀ = 10，选中 C 端口数据寄存器。
- A₁ A₀ = 11，选中 8255A 控制寄存器。

8255A 的端口编址与读写操作如表 11-1 所示。

表 11-1 8255A 端口编址与读写操作

A ₁	A ₀	\overline{RD}	\overline{WR}	\overline{CS}	输入操作(读)
0	0	0	1	0	从 A 端口读取数据
0	1	0	1	0	从 B 端口读取数据
1	0	0	1	0	从 C 端口读取数据
A ₁	A ₀	\overline{RD}	\overline{WR}	\overline{CS}	输出操作(写)
0	0	1	0	0	向 A 端口写入数据
0	1	1	0	0	向 B 端口写入数据
1	0	1	0	0	向 C 端口写入数据
1	1	1	0	0	向控制端口写入命令字

续表

A_1	A_0	\overline{RD}	\overline{WR}	\overline{CS}	无操作情况
×	×	×	×	1	总线悬浮
1	1	0	1	0	非法条件
×	×	1	1	0	总线悬浮

11.1.2 8255A 的控制字与初始化编程

8255A 的控制字有两个：方式选择控制字和 C 端口按位置 0/置 1 控制字。两个控制字共用一个端口地址，用特征位 D_7 位来区分。若 D_7 位=1，该控制字为方式选择控制字； D_7 位=0，该控制字为 C 端口按位置 0/置 1 控制字。8255A 的控制字须写入控制寄存器。

1. 方式选择控制字

方式选择控制字格式如图 11-3 所示。

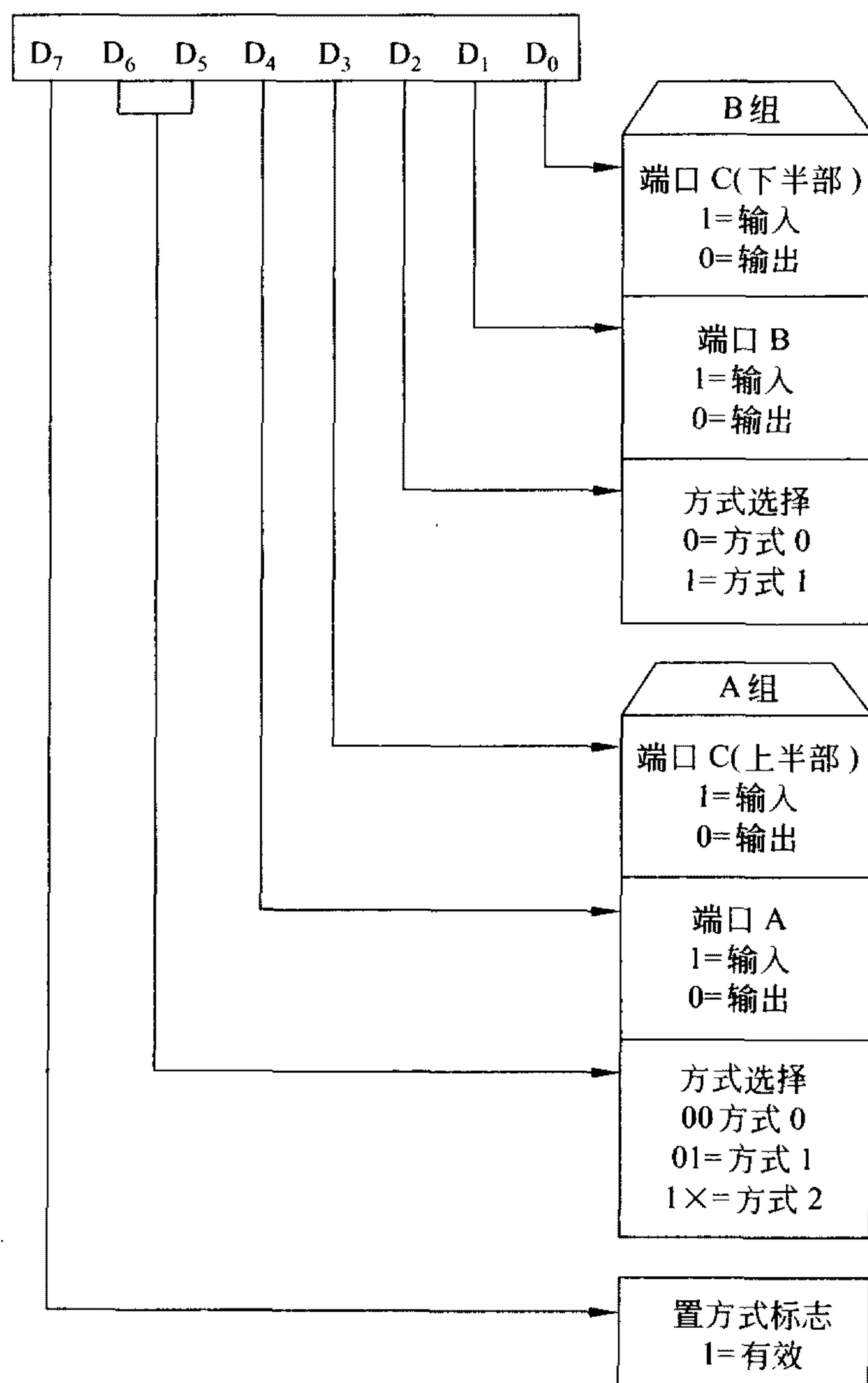


图 11-3 8255A 方式选择控制字

该控制字可以分别确定 A 端口和 B 端口的工作方式。C 端口分成两部分,C 端口的高四位 PC₇~PC₄ 随 A 端口,构成 A 组;C 端口的低四位 PC₃~PC₀ 随 B 端口,构成 B 组。

对于 A 端口和 B 端口而言,在设定工作方式时应该以八位为一个整体进行,而 C 端口高四位和低四位可以分别选择不同的输入/输出方式。

2. C 端口按位置 0/置 1 控制字

C 端口按位置 0/置 1 控制字的格式如图 11-4 所示。

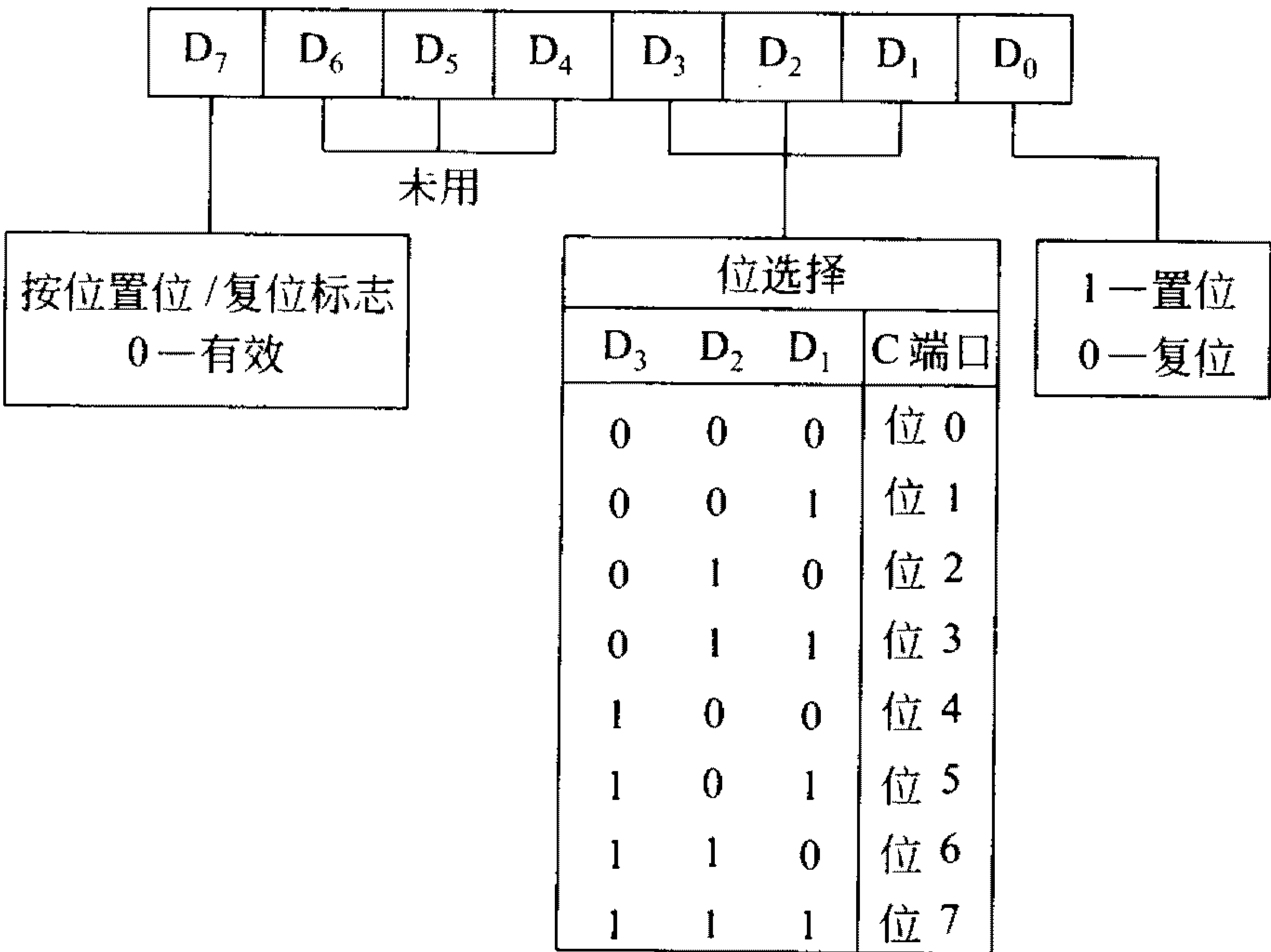


图 11-4 8255A C 端口按位置 0/置 1 控制字

C 端口的按位置 0/置 1 控制字尽管是对 C 端口进行操作,但该控制字必须写入控制端口,而不是写入 C 端口。

3. 8255A 初始化编程

8255A 初始化编程分两步进行: 首先把方式选择控制字写入控制端口,确定所用端口的工作方式;如果端口选择为方式 1 或方式 2,还要进一步明确,CPU 和 8255A 之间是用查询方式还是用中断方式交换信息,并以此来组织 C 端口置 0/置 1 控制字,写入 8255A 控制端口,使相应的中断允许标志(INTE)置 0 或置 1,从而达到禁止或开放中断的目的。

完成初始化编程后,CPU 可以用 IN、OUT 指令通过 8255A 和外设交换信息。

11.1.3 8255A 的工作方式

8255A 有三种工作方式:

- 方式 0: 基本型输入/输出方式。
- 方式 1: 选通型输入/输出方式。
- 方式 2: 双向数据传送方式。

A 端口可以工作在方式 0、方式 1、方式 2；B 端口可以工作在方式 0 和方式 1，不能工作在方式 2；C 端口可以工作在方式 0，不能工作在方式 1 和方式 2。

当 A 端口、B 端口工作在方式 1 或 A 端口工作在方式 2 时，C 端口配合 A 端口和 B 端口工作，为这两个端口的输入/输出提供联络信号。

1. 方式 0

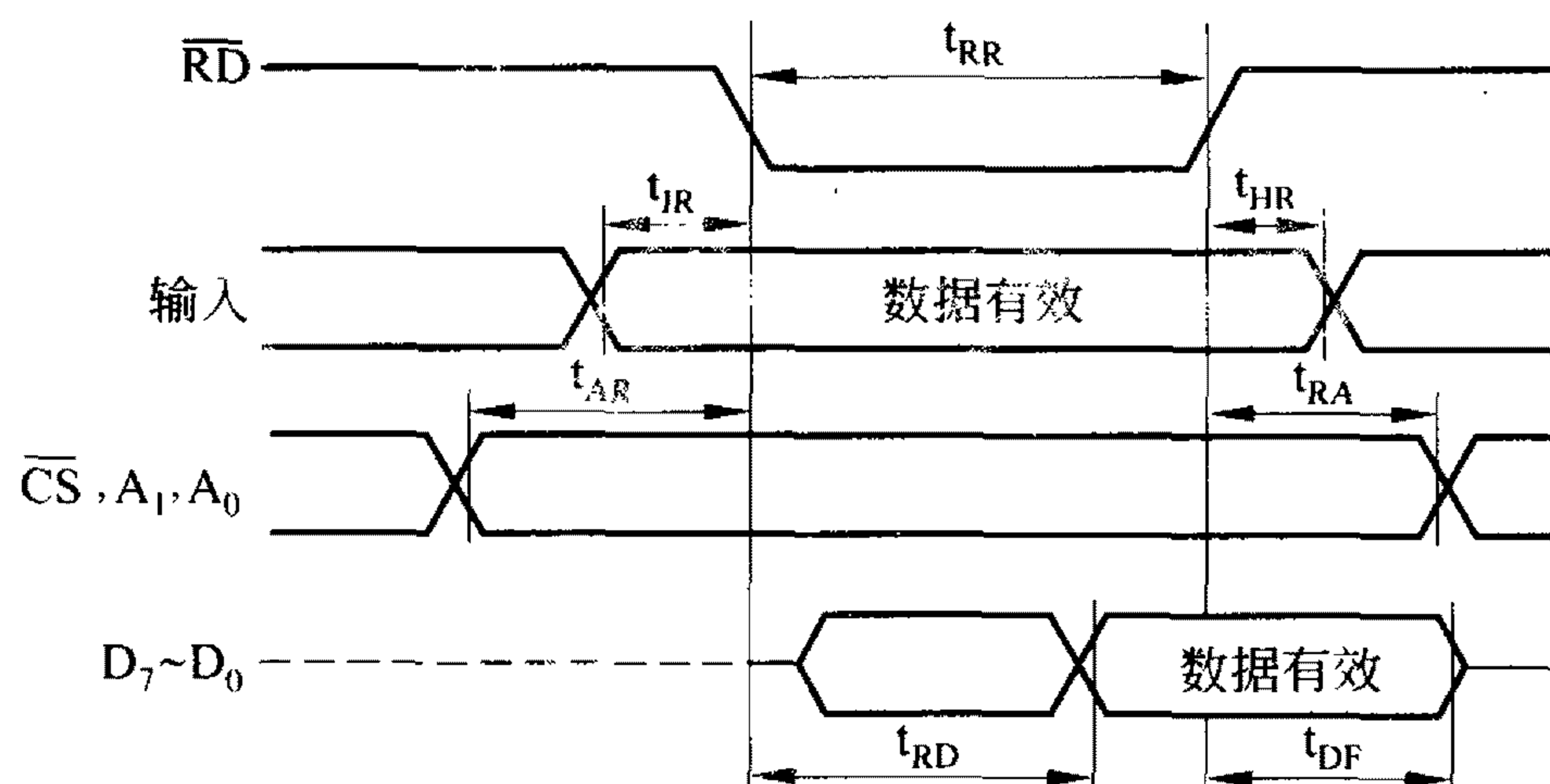
(1) 方式 0 的工作特点

方式 0 是基本型输入/输出方式，即无条件输入/输出方式。这时，端口和外设之间不需要联络信号。A 端口、B 端口和 C 端口可由方式选择控制字规定为输入或输出。

(2) 方式 0 的输入/输出时序

• 方式 0 的输入时序

图 11-5 为方式 0 的输入时序和参数说明。



符 号	参 数	8255A		单 位
		MIN	MAX	
t_{RR}	读脉冲宽度	300		ns
t_{IR}	输入领先于 \overline{RD} 的时间	0		ns
t_{HR}	输入滞后 \overline{RD} 的时间	0		ns
t_{AR}	地址稳定领先读信号的时间	0		ns
t_{RA}	读信号无效后地址保持时间	0		ns
t_{RD}	从读信号有效到数据稳定		250	ns
t_{DF}	读信号去除后至数据浮空	10	150	ns
t_{RY}	在两次读(或写)之间的时间间隔	850		ns

图 11-5 方式 0 的输入时序和参数说明

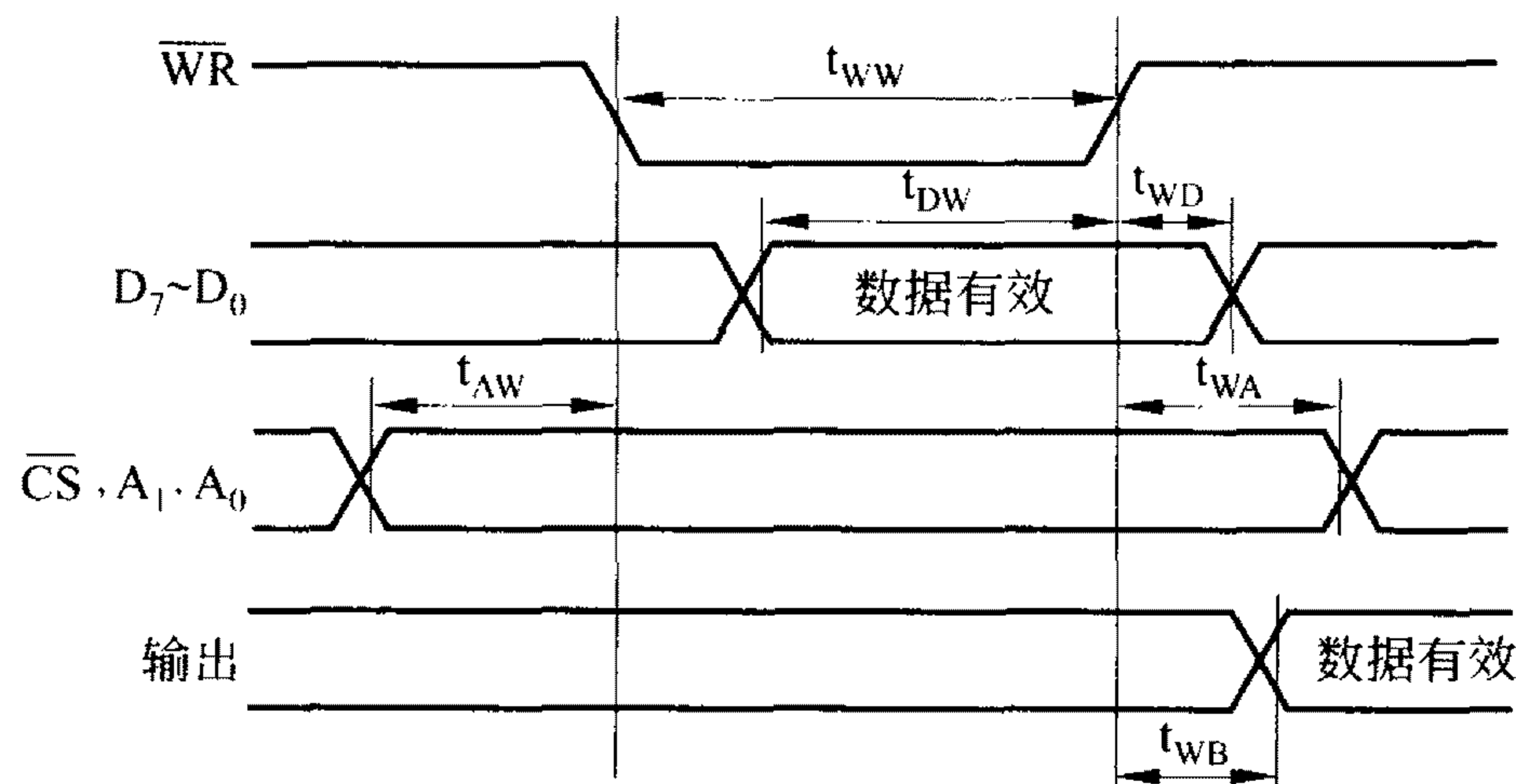
8255A 工作在方式 0 输入时，CPU 在读取数据之前，端口数据必须准备好。

当 CPU 对端口执行一条输入指令时， \overline{CS} 、 A_1 、 A_0 有效，8255A 被选中，随后 \overline{RD} 信号有效，读取端口数据，经 t_{RD} 时间延迟，端口数据被送到系统的数据总线上，完成一次输入操作。为了防止读出的数据出错， \overline{RD} 有效期间应保持地址信号有效，端口数据应保持到

读信号结束后才能消失,读脉冲的宽度不小于 300ns。

• 方式 0 的输出时序

图 11-6 为方式 0 的输出时序和参数说明。



符 号	参 数	8255A		单 位
		MIN	MAX	
t_{AW}	地址稳定领先写信号的时间	0		ns
t_{WA}	写信号后地址保持时间	20		ns
t_{WW}	写脉冲宽度	400		ns
t_{DW}	从写信号到数据有效	100		ns
t_{WD}	数据保持时间	30		ns
t_{WB}	从写信号结束到输出		350	ns

图 11-6 方式 0 的输出时序和参数说明

8255A 方式 0 输出是 CPU 将数据经数据总线,传送到 8255A 的端口数据线上。在 CPU 执行输出指令之前,端口数据线必须是空闲的。当 CPU 对端口执行一条输出指令时, \overline{CS} 、 A_1 、 A_0 有效,待输出的数据在系统数据线上,当 \overline{WR} 信号结束后,最长经过 t_{WB} ,端口数据线上就会出现有效数据, \overline{WR} 的宽度至少为 400ns,CPU 写入的数据在整个写操作期间要保持有效,当 \overline{WR} 结束后,还需至少保持 $t_{WD}=30\text{ns}$ 。

2. 方式 1

(1) 方式 1 的工作特点

方式 1 为选通型输入/输出方式。8255A 工作在方式 1 时,端口和外设之间必须有联络线,CPU 与 8255A 可以用查询方式或中断方式交换信息。

(2) 方式 1 输入

当 8255A 的 A 端口或 B 端口工作在方式 1 输入时,对应的联络信号如图 11-7 所示。

当 A 端口工作在方式 1 输入时, $PA_7 \sim PA_0$ 为端口的输入数据线, PC_5 和 PC_4 为联络线, PC_4 被自动定义为“输入”,改称为 \overline{STB}_A ; PC_5 被自动定义为“输出”,改称为 IBF_A , PC_3 被自动定义为中断请求输出线,改称为 $INTR_A$ 。 PC_5 、 PC_4 、 PC_3 不受方式选择命令

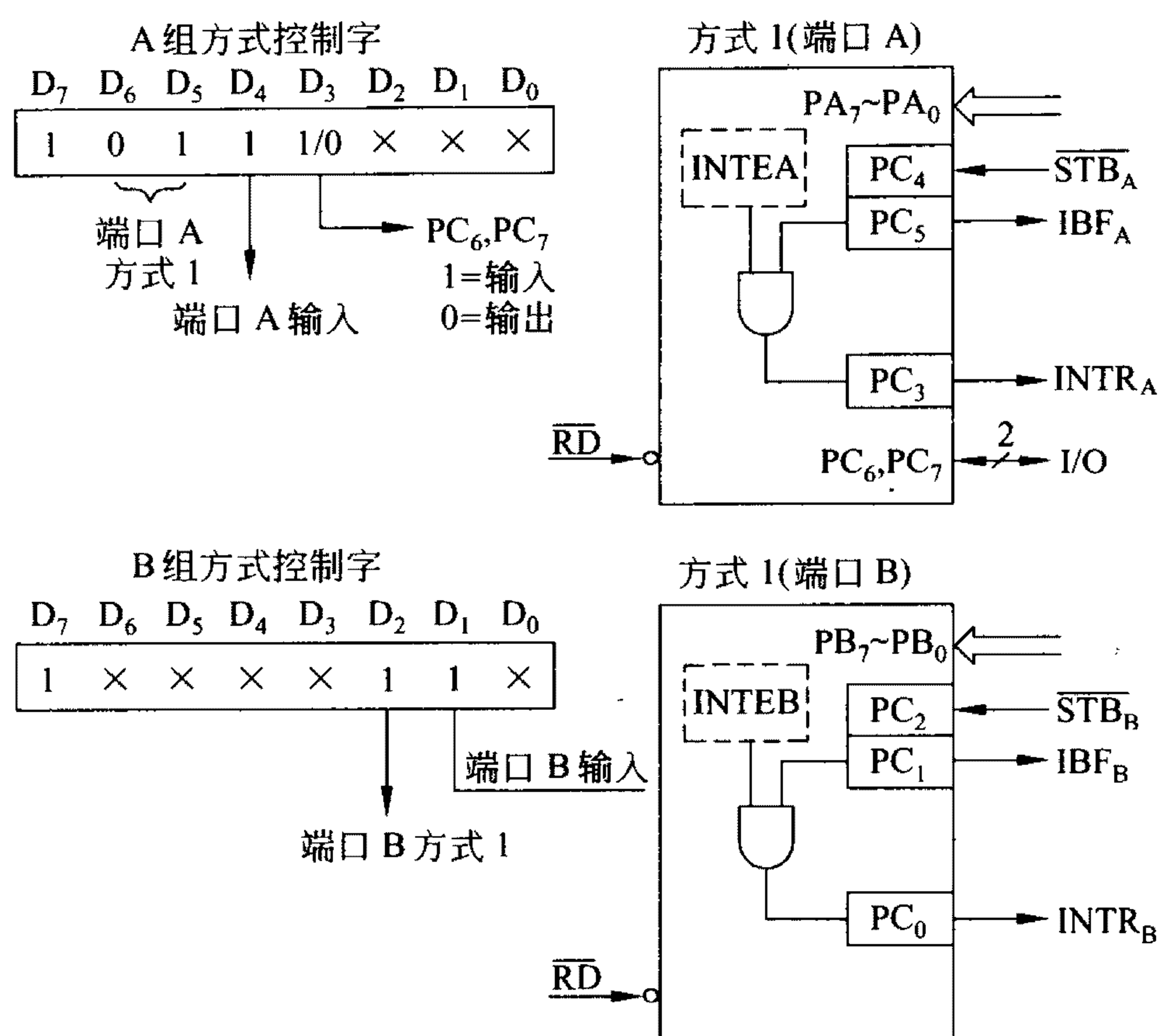


图 11-7 方式 1 输入时对应的联络信号

字的控制, PC_7 、 PC_6 空闲。

当 B 端口工作在方式 1 输入时, $PB_7 \sim PB_0$ 为端口的输入数据线。 PC_2 和 PC_1 为联络线, PC_2 被自动定义为“输入”, 改称为 \overline{STB}_B ; PC_1 被自动定义为“输出”, 改称为 IBF_B , PC_0 被自动定义为中断请求输出线, 改称为 $INTR_B$ 。 PC_2 、 PC_1 、 PC_0 不受方式选择命令字的控制。

端口联络线的功能如下:

\overline{STB} : 输入选通信号, 低电平有效, 由外设发往 8255A。 \overline{STB} 有效, 外设数据写入相应端口的输入缓冲器中。

IBF : 输入缓冲器满, 高电平有效, 由 8255A 发往外设。 $IBF=1$, 通知输入设备, 8255A 已经收到数据, 暂缓输入下一个数据。 CPU 采用查询方式从 8255A 读取数据之前, 应查询 IBF , 只有当 $IBF=1$ 时, CPU 才能从 A 端口或 B 端口读取输入数据。

$INTR$: 中断请求信号, 高电平有效。 在中断允许 ($INTEA=1$ 或 $INTEB=1$) 的前提下, 8255A 接收到一个端口数据后 ($IBF=1$), 向 CPU 发出中断请求。

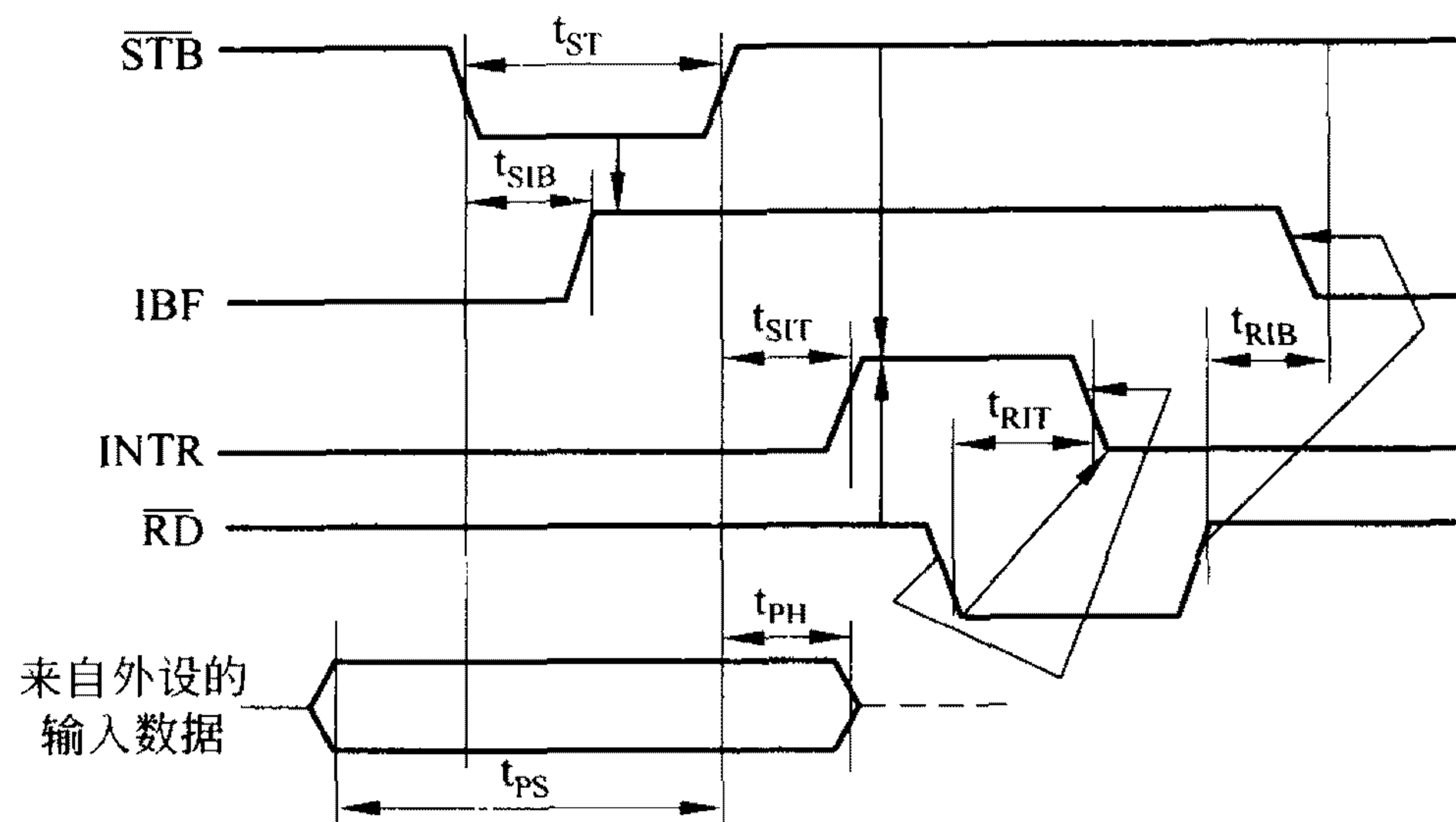
$INTEA$: A 端口中断允许寄存器, 受 PC_4 的置 0/置 1 命令字控制, 当 $PC_4=1$ 时, A 端口允许中断。

$INTEB$: B 端口中断允许寄存器, 受 PC_2 的置 0/置 1 命令字控制, 当 $PC_2=1$ 时, B 端口允许中断。

(3) 方式 1 的输入时序

方式 1 的输入时序如图 11-8 所示。

方式 1 的输入过程是从外设输入数据并发出 \overline{STB} 有效信号开始。 8255A 要求选通脉冲 \overline{STB} 的宽度 t_{ST} 大于 500ns。 在 \overline{STB} 下降沿之后, 经过约 t_{SIB} 时间, 8255A 接收到数据,



符 号	参 数	8255A		单 位
		MIN	MAX	
t_{ST}	\overline{STB} 脉冲宽度	500		ns
t_{SIB}	$\overline{STB}=0$ 到 $IBF=1$		300	ns
t_{SIT}	$\overline{STB}=1$ 到 $INTR=1$		300	ns
t_{RIB}	$\overline{RD}=1$ 到 $IBF=0$		300	ns
t_{RIT}	$\overline{RD}=0$ 到 $INTR=0$		400	ns
t_{PS}	数据提前 \overline{STB} 无效的时间	0		ns
t_{PH}	数据保持时间	180		ns

图 11-8 方式 1 的输入时序

IBF 变为高电平，表示输入缓冲器满。当 \overline{STB} 变为高电平，如果中断允许($INTE=1$)， \overline{STB} 的上升沿经过 t_{SIT} 时间后 INTR 有效，向 CPU 发出中断申请。CPU 响应中断，用 IN 指令读取数据，产生 \overline{RD} 信号。 \overline{RD} 信号变低后经过 t_{RIT} 时间，INTR 变为无效，撤销本次中断请求。 \overline{RD} 信号的上升沿经过 t_{RIB} 时间，IBF 变为低电平，表示输入缓冲器为空，从而结束方式 1 的输入过程。

(4) 方式 1 输出

当 A 端口或 B 端口工作于方式 1 输出时，对应的联络信号如图 11-9 所示。

当 A 端口工作于方式 1 输出时， PC_7 自动定义为输出线，改称 \overline{OBF}_A ， PC_6 自动定义为输入线，改称 \overline{ACK}_A 。 \overline{OBF}_A 和 \overline{ACK}_A 为一对联络信号。

当 B 端口工作于方式 1 输出时， PC_1 自动定义为输出线，改称 \overline{OBF}_B ， PC_2 自动定义为输入线，改称 \overline{ACK}_B 。 \overline{OBF}_B 和 \overline{ACK}_B 为一对联络信号。

端口联络线的功能如下：

\overline{OBF} ：输出缓冲器满，低电平有效。 \overline{OBF} 为低电平，表示 CPU 已将输出数据写入指定的端口数据寄存器中，当 CPU 用查询方式向 8255A 输出数据时，应先查询 \overline{OBF} ，只有当 $\overline{OBF}=1$ 时，CPU 才能输出下一个数据。

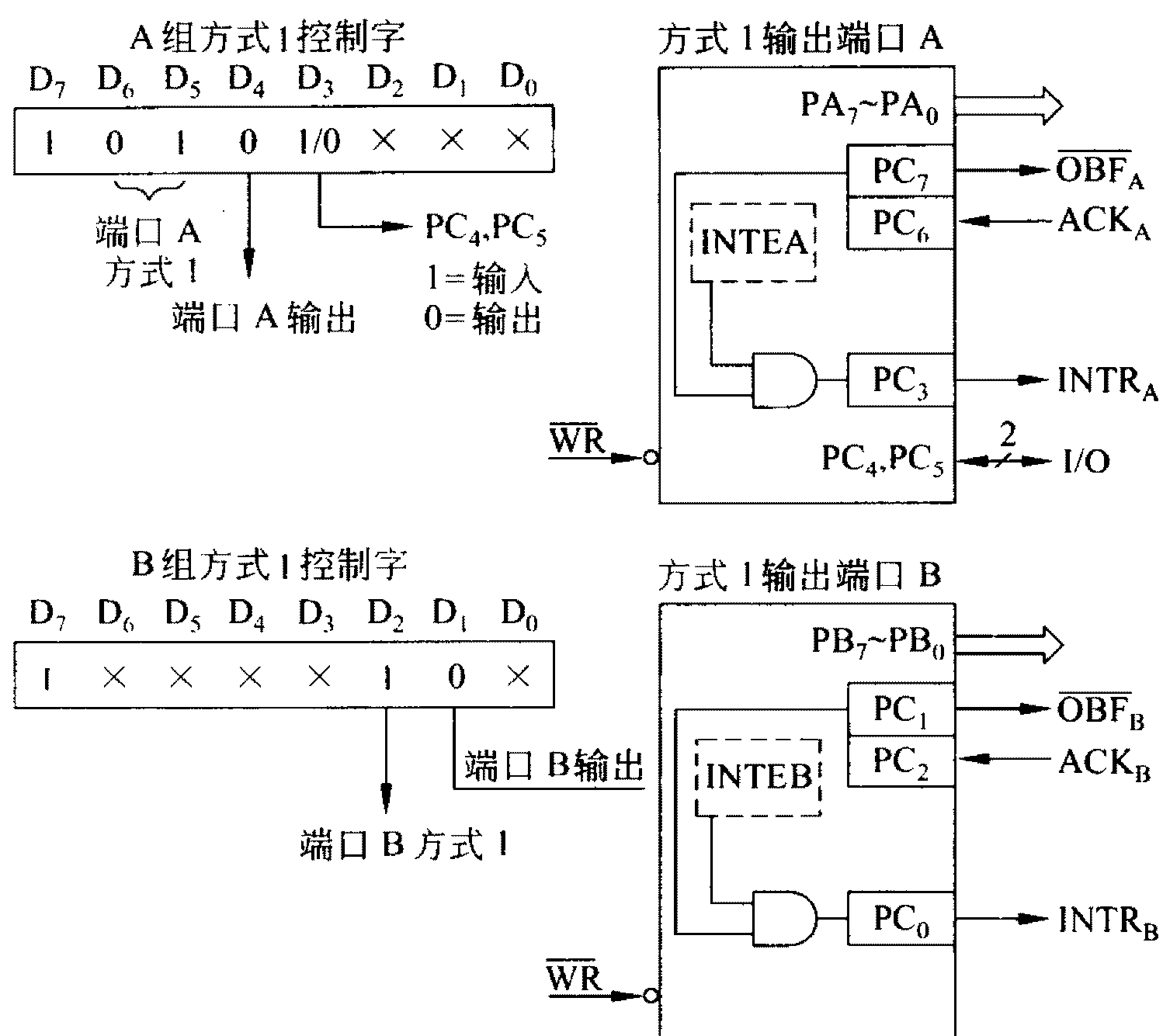


图 11-9 方式 1 输出时对应的联络信号

$\overline{\text{ACK}}$: 外设的应答信号,低电平有效。该信号是已接收数据的外设对 $\overline{\text{OBF}}$ 的应答信号。8255A 规定: 外设取走端口数据后,必须向 $\overline{\text{ACK}}$ 端子发送脉宽大于 300ns 的负脉冲。

INTR: 中断请求信号,高电平有效。在中断允许(INTEA=1 或 INTEB=1)的前提下,当外设取走端口数据之后($\overline{\text{OBF}}=1$),向 CPU 发出中断请求。

INTEA: 受 PC₆ 的置 0/置 1 命令字控制,当 PC₆=1 时,A 端口允许中断。

INTEB: 受 PC₂ 的置 0/置 1 命令字控制,当 PC₂=1 时,B 端口允许中断。

(5) 方式 1 的输出时序

方式 1 的输出时序如图 11-10 所示。

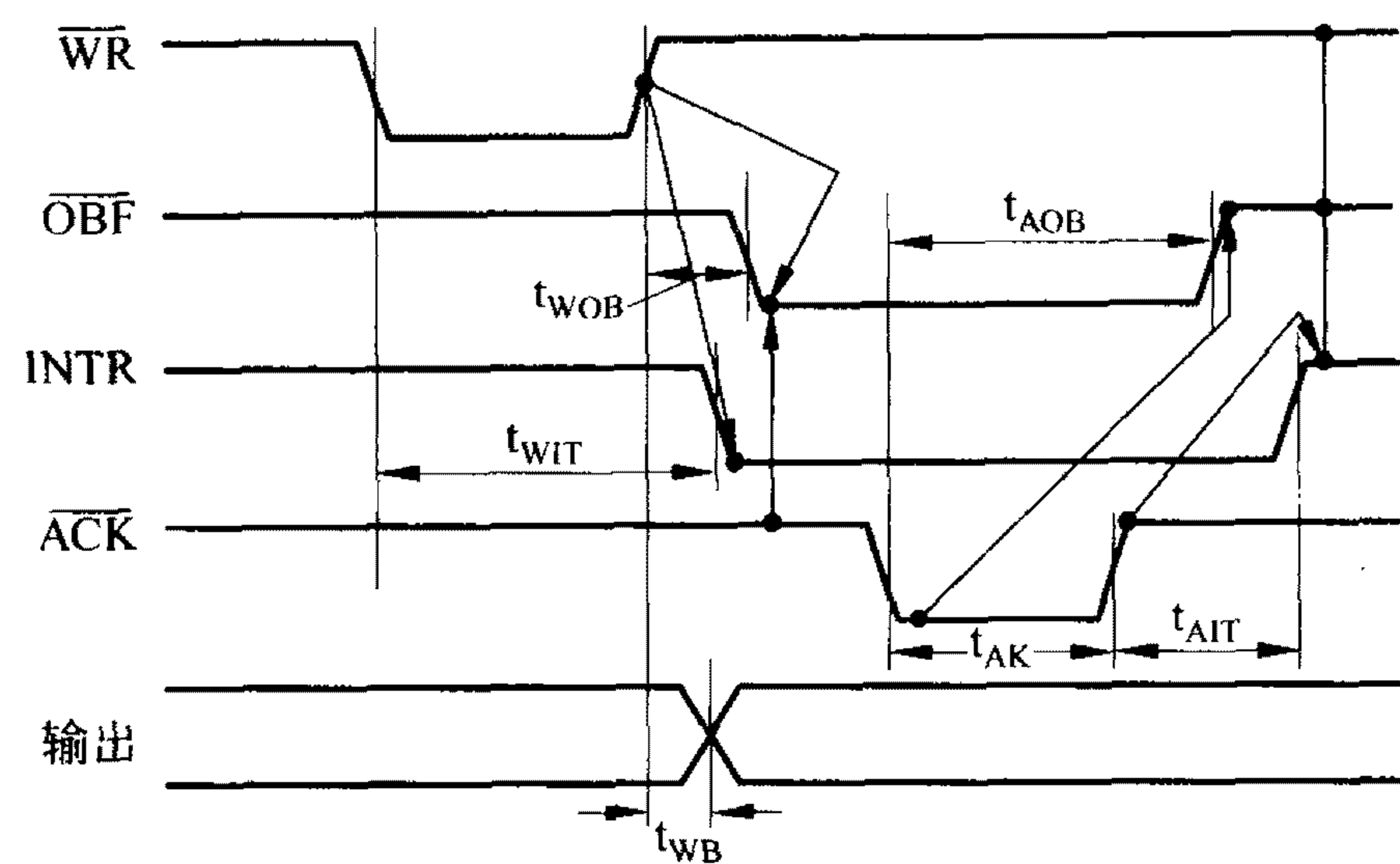
CPU 采用中断方式和 8255A 交换信息时,输出过程是由 CPU 响应中断开始的。CPU 响应中断后,执行 OUT 指令输出数据, $\overline{\text{WR}}$ 信号有效, $\overline{\text{WR}}$ 信号的上升沿一方面使中断请求信号变为无效,表示 CPU 已响应中断;另一方面,使 $\overline{\text{OBF}}$ 信号有效,表示输出缓冲器满,通知外设取走数据。外设从端口取走数据后,发出 $\overline{\text{ACK}}$ 应答信号。 $\overline{\text{ACK}}$ 信号有效后约 350ns, $\overline{\text{OBF}}$ 信号变为高电平,表示输出缓冲器空, $\overline{\text{ACK}}$ 信号的宽度应大于 300ns。 $\overline{\text{ACK}}$ 信号无效后约 350ns,INTR 信号变为有效,向 CPU 发出中断申请,一个新的输出过程开始了。

3. 方式 2

(1) 方式 2 的工作特点

方式 2 为双向传输方式,只有 A 端口可工作在方式 2。

当 A 端口工作于方式 2 时,A 端口为输入输出双向端口,PA₇~PA₀ 为双向数据线,PC₄和PC₅为一对输入联络线,PC₆和PC₇为一对输出联络线,PC₃为中断请求线。当A端口



符 号	参 数	8255A		单 位
		MIN	MAX	
t_{WOB}	$\overline{WR}=1$ 到 $\overline{OBF}=0$		650	ns
t_{WIT}	$\overline{WR}=0$ 到 $INTR=0$		850	ns
t_{AOB}	$\overline{ACK}=0$ 到 $\overline{OBF}=1$		350	ns
t_{AK}	\overline{ACK} 脉冲宽度	300		ns
t_{AIT}	$\overline{ACK}=1$ 到 $INTR=1$		350	ns
t_{WB}	$\overline{WR}=1$ 到 输出		350	ns

图 11-10 方式 1 的输出时序

工作于方式 2 时,B 端口可以工作在方式 0 或方式 1。

(2) 方式 2 工作时的联络信号

图 11-11 给出了 8255A 工作于方式 2 时的联络信号。

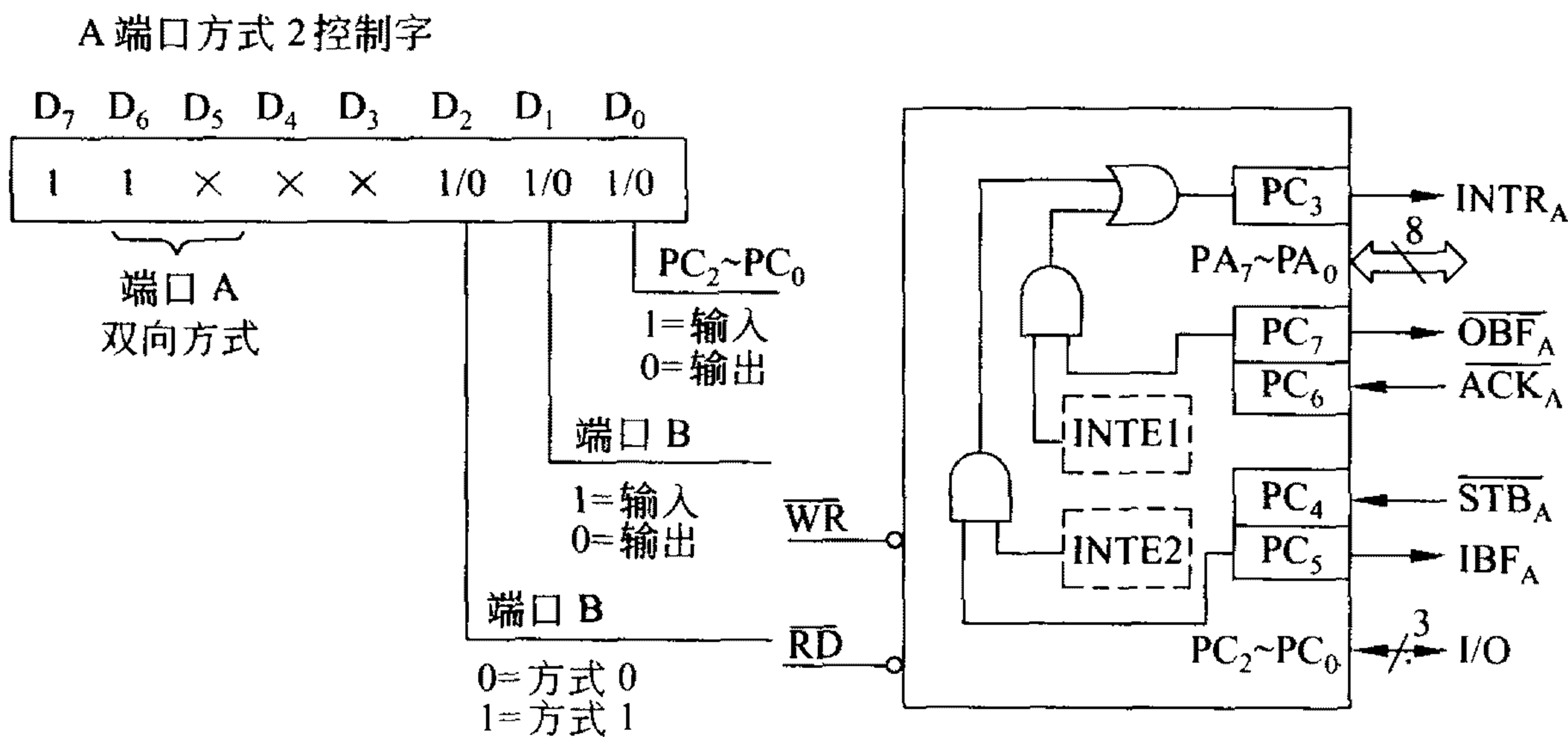


图 11-11 8255A 工作于方式 2 时联络信号

如图 11-11 所示,当 A 端口工作于方式 2 时。PC₇ 自动定义为输出线,改称 \overline{OBF}_A , PC₆ 自动定义为输入线,改称 \overline{ACK}_A ,PC₅ 自动定义为输出线,改称 IBF_A ,PC₄ 自动定义为输入线,改称 \overline{STB}_A ,PC₃ 自动定义为输出线,改称 $INTR_A$ 。

各联络信号功能如下：

INTR_A ：中断请求信号，高电平有效。

$\overline{\text{STB}}_A$ ：输入选通信号，低电平有效。 $\overline{\text{STB}}_A$ 有效，外设输入的数据存入 A 端口。

IBF_A ：输入缓冲器满，高电平有效。 IBF_A 有效，表示 A 端口已有数据，外设应暂缓输入新的数据。

$\overline{\text{OBF}}_A$ ：输出缓冲器满，低电平有效。 $\overline{\text{OBF}}_A$ 有效，表示 CPU 已将数据写入 A 端口，外设可以取走数据。

$\overline{\text{ACK}}_A$ ：外设的应答信号，低电平有效。 $\overline{\text{ACK}}_A$ 有效，表示 A 端口输出的数据已被外设取走。

INTE1 ：A 端口“输出中断允许”寄存器，受 PC_6 的置 0/置 1 命令字控制。

INTE2 ：A 端口“输入中断允许”寄存器，受 PC_4 的置 0/置 1 命令字控制。

(3) 方式 2 的工作时序

方式 2 的时序相当于方式 1 的输入时序和输出时序的组合，如图 11-12 所示。

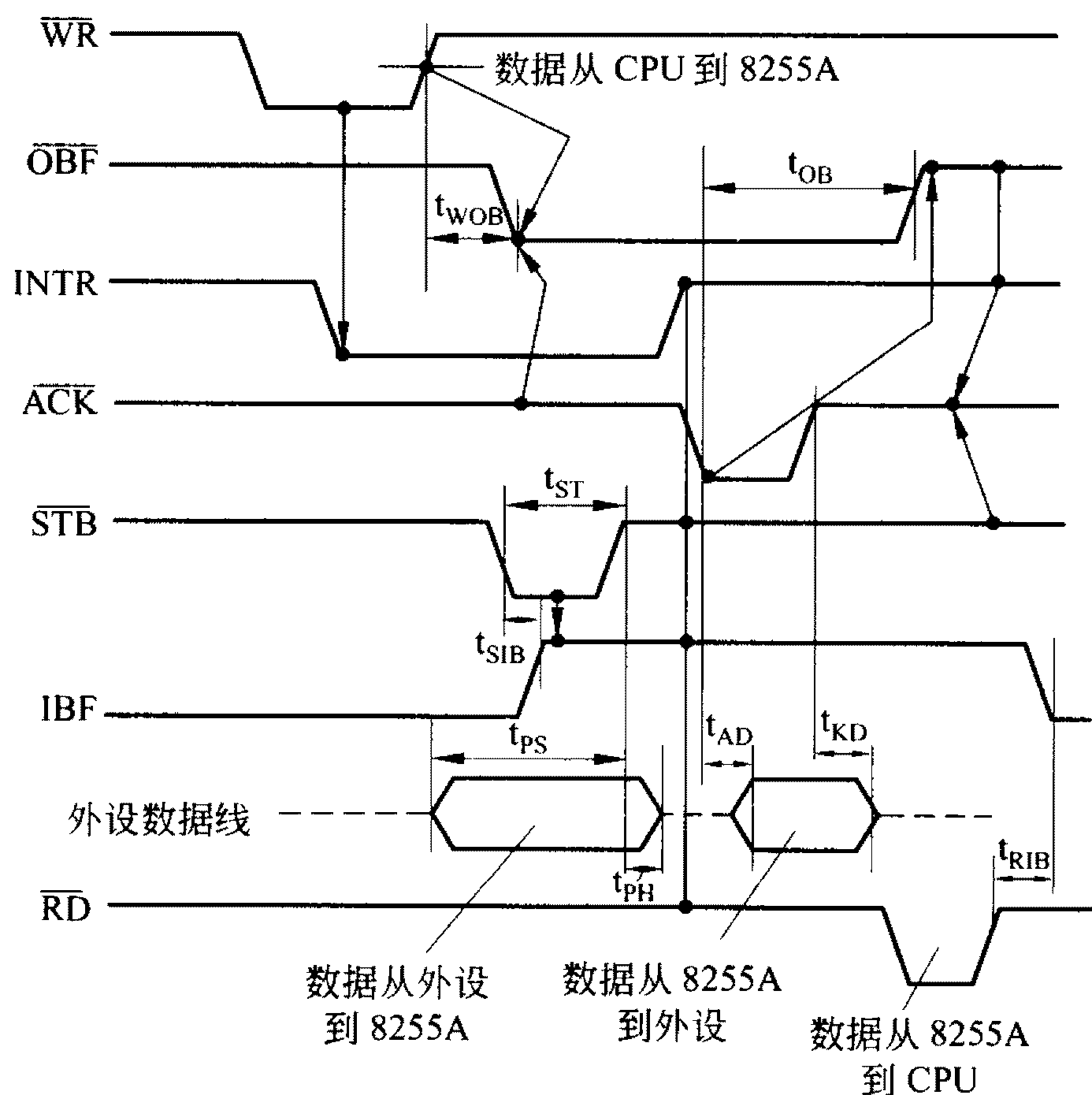


图 11-12 方式 2 时序图

CPU 执行一条针对 A 端口的输出指令引发输出过程。 $\overline{\text{WR}}$ 信号使 INTR 信号变为无效， $\overline{\text{WR}}$ 的上升沿使 $\overline{\text{OBF}}_A$ 有效，表示输出缓冲器满，通知外设可将端口数据取走。外设取走数据后，向 8255A 发出应答信号 $\overline{\text{ACK}}_A$ ， $\overline{\text{ACK}}_A$ 的有效使 $\overline{\text{OBF}}_A$ 复位，可以继续输出数据。

选通信号 $\overline{\text{STB}}_A$ 引发输入过程，选通信号有效，将输入数据锁存到 A 端口的输入锁存器中， IBF_A 变为高电平，表示输入缓冲器满。选通信号 $\overline{\text{STB}}_A$ 结束时，中断请求信号变为高电平。CPU 响应中断进行读操作， $\overline{\text{RD}}$ 信号有效，将数据从 A 端口读到 CPU 中， IBF_A 和中断请求信号变为低电平，数据输入过程结束。

11.2 8255A 应用

1. 8255A 在微型计算机系统中的应用

早期的 PC/XT 微型计算机系统(8088 CPU)使用一片 8255A,系统分配的端口地址为 A 端口 60H,B 端口 61H,C 端口 62H,控制端口 63H。

A 端口工作在基本型输入方式,用来暂存键盘接口电路串→并转换后的按键扫描码。B 端口工作在基本型输出方式,PB₇、PB₆ 控制键盘接口电路,PB₁、PB₀ 控制发声系统。C 端口工作在基本型输入方式,存放“系统配置开关”的信息。

80286 以后的微型计算机系统,系统配置由软件完成,8255A 的功能被其他多功能芯片取代,为了保持和低档微型计算机的兼容性,系统仍使用 8255A 的端口地址,从应用的角度讲,仍然可以从 60H 端口地址读取按键扫描码,仍然可以使用 PB₁、PB₀ 控制发声系统。

2. 8255A 应用举例

8255A 为可编程芯片,工作时应首先对其进行初始化编程,初始化编程分两步进行。

首先应根据要求,正确地选择端口工作方式,把方式选择控制字写入 8255A 的控制端口,如果所用的端口选择为方式 1 或方式 2,还要进一步明确,CPU 和 8255A 之间是用查询方式还是用中断方式交换信息,并以此来组织 C 端口置 0/置 1 控制字,写入 8255A 控制端口,使相应的中断允许寄存器(INTEA、INTEB 或者 INTE1、INTE2)置 0 或置 1,从而达到禁止或开放中断的目的。完成了初始化编程之后,CPU 就可以用 IN、OUT 指令通过 8255A 和外设交换数据了。

【例 11.2.1】 设系统机外扩了一片 8255A 以及相应的实验电路,如图 11-13 所示。

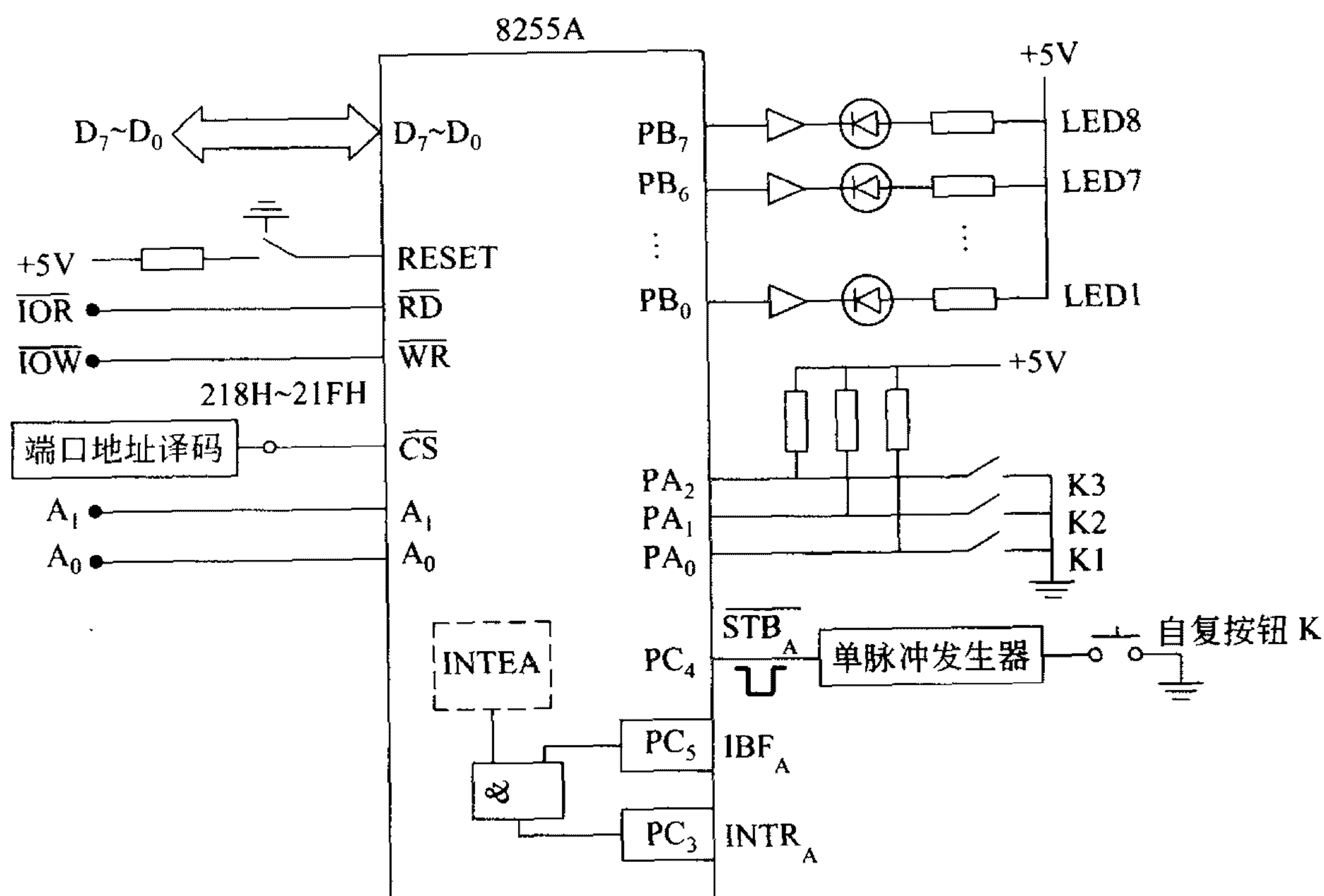


图 11-13 外扩 8255A 实验电路

要求：先预置开关 $K_3 \sim K_1$ 为一组状态，然后按下自复按钮 K 产生一个负脉冲信号输入到 PC_4 。用发光二极管 LED_i 亮，显示 $K_3 \sim K_1$ 的状态。主机键盘有任意键按下时结束演示。

要求：	$K_3 K_2 K_1 = 000$ 时，LED1 亮	$K_3 K_2 K_1 = 001$ 时，LED2 亮
	$K_3 K_2 K_1 = 010$ 时，LED3 亮	$K_3 K_2 K_1 = 011$ 时，LED4 亮
	$K_3 K_2 K_1 = 100$ 时，LED5 亮	$K_3 K_2 K_1 = 101$ 时，LED6 亮
	$K_3 K_2 K_1 = 110$ 时，LED7 亮	$K_3 K_2 K_1 = 111$ 时，LED8 亮

$K_3 \sim K_1$ 闭合为 0，断开为 1。

【设计思路】

(1) 8255A 端口地址

分析电路可知：当端口地址为 $218H \sim 21FH$ 时选中 8255A，再由地址线 A_1, A_0 选择 8255A 内部寄存器，所以，8255A 内部寄存器端口地址应为：

A 端口数据寄存器：218H, 21CH

B 端口数据寄存器：219H, 21DH

C 端口数据寄存器：21AH, 21EH

控制寄存器：21BH, 21FH

(2) 8255A 工作方式的选择

分析电路可知，当 PC_4 (即 $\overline{STB_A}$) 接收到负脉冲信号之后， $K_3 \sim K_1$ 的状态信息被锁存到 A 端口数据寄存器。CPU 应读取 A 端口信息，再根据 $PA_2 \sim PA_0$ 的状态，输出相应数据到 B 端口，仅仅使 LED_i 亮。为此，8255A A 端口应工作在选通型输入，B 端口应工作在基本型输出。

(3) CPU 与 8255A 交换信息的方式

CPU 何时执行输入指令，读取 A 端口信息呢？有两种方式可供选择，即查询方式和中断方式。

① 查询方式 当 PC_4 收到负脉冲之后， $K_3 \sim K_1$ 的状态信息被存入 A 端口数据寄存器，8255A 使“输入缓冲器满”标志线 IBF_A (即 PC_5) 置 1。因此程序可以查询 PC_5 。当 PC_5 为 1 时，再执行 A 端口输入指令。

注意，查询 $\overline{STB_A}$ (即 PC_4) 是不规范的操作，因为 8255A 允许 $\overline{STB_A}$ 的负脉宽小到 500ns。程序查询这样窄的脉冲有可能错过机会。

② 中断方式 采用中断方式和 8255A 交换信息时，使用系统用户中断。图 11-13 的实验电路需要补接一根线，把 8255A PC_3 连到系统总线端子 B_4 。事先使 A 端口中断允许寄存器 $INTEA$ 为 1 (即令 $PC_4 = 1$)，那么，当 PC_4 收到负脉冲之后， $K_3 \sim K_1$ 的状态信息被存入 A 端口数据寄存器，8255A 使 PC_3 为 1，提出中断请求。程序员在服务程序中安排一条输入指令，读取 A 端口信息……。

(4) 初始化参数的计算

用查询方式编程，8255A 方式选择命令字为 $B0H$ ，A 端口禁止中断的命令字为 $08H$ 。用中断方式编程，8255A 方式选择命令字为 $B0H$ ，A 端口允许中断的命令字为 $09H$ 。

【查询方式程序清单】

```

;FILENAME: 1121_1. ASM
DATA SEGMENT
MSG DB '8255A READY...',0DH,0AH,'$'
TAB DB 11111110B
DB 11111101B
DB 11111011B
DB 11110111B
DB 11101111B
DB 11011111B
DB 10111111B
DB 01111111B
DATA ENDS
CODE SEGMENT
ASSUME CS: CODE,DS: DATA
BEG: MOV AX,DATA
MOV DS,AX
CALL I8255A ;8255A 初始化
MOV AH,9
MOV DX,OFFSET MSG
INT 21H ;给出操作提示
SCAN: MOV AH,1
INT 16H ;有键入?
JNZ RETURN ;有
MOV DX,21AH
IN AL,DX ;读 8255A C 端口
TEST AL,00100000B ;PC5 = 1?
JZ SCAN ;NO
MOV DX,218H
IN AL,DX ;读 8255A A 端口
AND AL,07H
MOV BX,OFFSET TAB
XLAT TAB ;查表
MOV DX,219H
OUT DX,AL ;表项输出到 B 端口
JMP SCAN
RETURN: MOV AH,4CH
INT 21H ;返回 DOS
I8255A PROC
MOV DX,21BH
MOV AL,0B0H
OUT DX,AL ;写入工作方式字
MOV AL,08H

```

```

                                OUT      DX,AL          ;令 PC4=0 (INTE A =0)
                                MOV      DX,219H
                                MOV      AL,0FFH
                                OUT      DX,AL          ;熄灭 LED
                                RET
I8255A  ENDP
CODE    ENDS
                                END      BEG

```

【中断方式程序清单】

```

                                ;FILENAME: 1121_2. ASM
DATA    SEGMENT
MSG     DB      '8255A READY...',0DH,0AH,'$'
TAB     DB      11111110B
        DB      11111101B
        DB      11111011B
        DB      11110111B
        DB      11101111B
        DB      11011111B
        DB      10111111B
        DB      01111111B
DATA    ENDS
CODE    SEGMENT
        ASSUME  CS: CODE,DS: DATA
BEG:    MOV      AX,DATA
        MOV      DS,AX
        CLI
        CALL     I8255A          ;8255A 初始化
        CALL     WRITE0A        ;置换 0AH 型中断向量
        CALL     I8259          ;开放用户中断
        MOV      AH,9
        MOV      DX,OFFSET MSG
        INT      21H            ;给出操作提示
        STI          ;开中断
SCAN:   MOV      AH,1
        INT      16H            ;有键入?
        JZ       SCAN          ;无,转移
        IN       AL,0A1H
        OR       AL,000000010B
        OUT      0A1H,AL        ;屏蔽用户中断
        MOV      AH,4CH
        INT      21H            ;返回 DOS
SERVICE PROC
        PUSH     AX

```

```

PUSH        DS
MOV         AX,DATA
MOV         DS,AX
MOV         DX,218H
IN          AL,DX                ;读 8255A A 端口
AND         AL,07H
MOV         BX,OFFSET TAB
XLAT        TAB                  ;查表
MOV         DX,219H
OUT         DX,AL                ;表项送 8255A B 端口
MOV         AL,20H              ;中断结束命令
OUT         20H,AL              ;→主 8259
POP         DS
POP         AX
IRET                        ;中断返回
SERVICE    ENDP
I8255A      PROC
MOV         DX,21BH
MOV         AL,0B0H
OUT         DX,AL                ;写入方式字
MOV         AL,09H
OUT         DX,AL                ;PC4 = 1, (INTE A = 1)
MOV         DX,219H
MOV         AL,0FFH
OUT         DX,AL                ;熄灭 LED
RET
I8255A      ENDP
WRITE0A     PROC
PUSH        DS
MOV         AX,CODE
MOV         DS,AX
MOV         DX,OFFSET SERVICE
MOV         AX,250AH
INT         21H
POP         DS
RET
WRITE0A     ENDP
I8259       PROC
IN          AL,21H
AND         AL,11111011B
OUT         21H,AL                ;开放从 8259 中断
IN          AL,0A1H
AND         AL,11111101B
OUT         0A1H,AL              ;开放用户中断

```



```

                RET
18259          ENDP
CODE          ENDS
                END
                BEG

```

【例 11.2.2】 测试 8254 工作方式，验证计数过程。

8254 每一个通道有 6 种工作方式，各种方式的计数过程也不一样，其中方式 3 的计数过程较为特殊，本例通过实验测试 8254 的方式 3，验证方式 3 的计数过程，图 11-14 是实验电路。

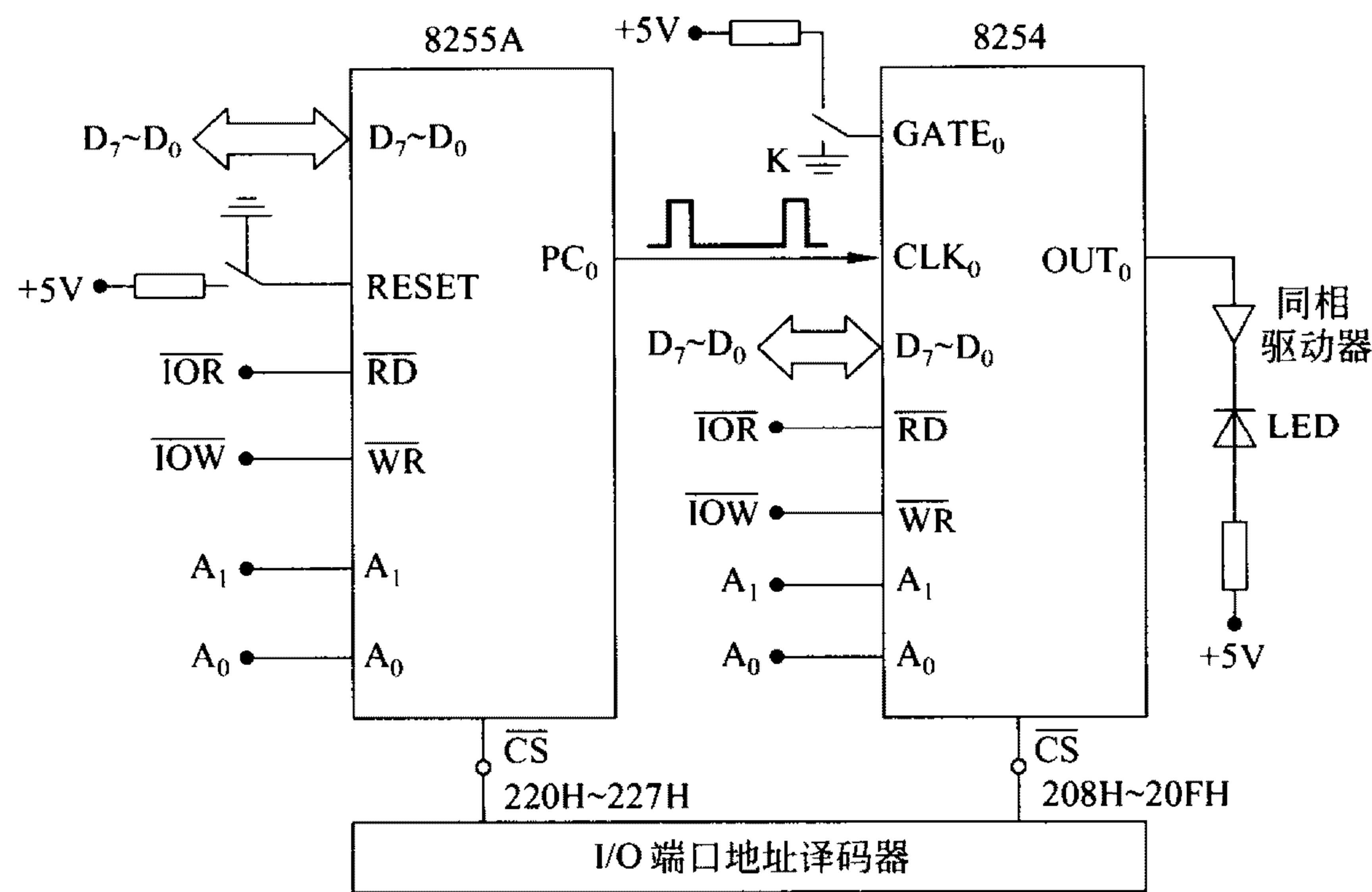


图 11-14 【例 11.2.2】实验电路

【实验电路】

假设微型计算机系统外扩了一片 8255A、一片 8254 及 I/O 端口地址译码器。8254 GATE₀ 通过开关 K，接至 +5V 或地，OUT₀ 外接一个 LED 显示电路，当 OUT₀ 为低电平时 LED 亮(OUT₀ 输出用示波器监测也可以)。用 8255A PC₀ 输出的正脉冲，做为 8254 的 0 号计数器的计数脉冲。

【程序清单】

```

;FILENAME: 1122.ASM
DISP      MACRO      REG,NN
LOCAL     LAST1, LAST2, NEXT
MOV       CH, NN/4
LAST1:    MOV         CL, 4
LAST2:    MOV         AL, '0'
           ROL         REG, 1
           JNC         NEXT
           MOV         AL, '1'
NEXT:     MOV         AH, 0EH

```

```

INT      10H
DEC      CL
JNZ      LAST2
MOV      AL,' '
INT      10H
DEC      CH
JNZ      LAST1
MOV      AL,0DH
INT      10H
MOV      AL,0AH
INT      10H
ENDM
;-----
CODE      SEGMENT
ASSUME    CS: CODE
CON_8255A EQU    223H      ;8255A 控制端口地址
CON_8254   EQU    20BH      ;8254 控制端口地址
CH0_8254   EQU    208H      ;8254 CH0 数据端口
BEG:       CALL    I8255A    ;8255A 初始化
           MOV     AL,00110110B ;0 号计数器方式 3
           MOV     DX,CON_8254 ;读/写方式: 先低 8 位后高 8 位
           OUT     DX,AL      ;命令字→8254
LAST:      MOV     AX,4       ;计数初值=4
WRITE:     MOV     DX,CH0_8254
           OUT     DX,AL      ;写入低 8 位计数初值
           MOV     AL,AH
           OUT     DX,AL      ;写入高 8 位计数初值
WAIT_IN:   MOV     AH,1
           INT     21H        ;等待键入
           CMP     AL,1BH
           JE      EXIT       ;是'Esc',转移
           CMP     AL,'4'
           JE      LAST       ;是'4',转移
           CMP     AL,'5'
           JNE     NEXT       ;不是'5',转移
           MOV     AX,5       ;计数初值=5
           JMP     WRITE
NEXT:      CALL    PC0        ;PC0 输出正脉冲→CLK0
           MOV     DX,CON_8254
           MOV     AL,11000010B ;读出命令字→AL
           OUT     DX,AL      ;锁存 0 号计数器状态和计数值
           MOV     DX,CH0_8254
           IN      AL,DX       ;读状态字
           MOV     BL,AL       ;→BL

```

```

                                DISP      BL,8                ;显示状态信息
                                IN         AL,DX              ;读低 8 位计数值
                                MOV        BL,AL              ;→BL
                                IN         AL,DX              ;读高 8 位计数值
                                MOV        BH,AL              ;→BH
                                DISP      BX,16               ;显示当前计数值
                                JMP        WAIT_IN
EXIT:  MOV        AH,4CH
                                INT       21H

;-----
I8255A  PROC                ; 8255A 初始化
                                MOV        AL,80H
                                MOV        DX,CON_8255A
                                OUT        DX,AL              ;预置 C 端口方式 0 输出
                                MOV        AL,0
                                OUT        DX,AL              ;PC0=0
                                RET
PC0:   MOV        AL,1
                                MOV        DX,CON_8255A
                                OUT        DX,AL              ;PC0=1
                                MOV        AL,0
                                OUT        DX,AL              ;PC0=0
                                RET
I8255A  ENDP
CODE    ENDS
                                END        BEG

```

【程序分析与实验方法】

① 程序对 8255A 初始化,令 C 端口工作在方式 0 输出。对 8254 初始化,使 0 号计数器工作在方式 3,二进制计数方式。程序可以向 0 号计数器写入两种计数初值,即 4 或 5。实验过程中,按下数字键“4”,则装入的初值为 4,按下数字键“5”,则装入初值 5。程序执行时,将自动装入计数初值 4。

② 实验时将开关 K 上合,使 GATE₀ 为 +5V,0 号计数器能正常工作,然后调入程序运行。

程序运行后测试人员首先键入 4 或 5,然后,每按动一次回车键,就使 8255A PC₀ 发出一个正脉冲,该脉冲使 8254 的 0 号计数器进行一次计数操作,随后程序就锁存 0 号计数器的计数值,然后读出当前计数值送屏幕显示,按 Esc 键测试结束。

③ 实验结果表明:当计数初值为偶数的时候,CLK₀ 每输入一个脉冲,计数值减 2,当计数值减到 0 的时候,内部完成初值的重装,OUT₀ 输出低电平时点亮 LED,正负脉冲的宽度为 1:1。

④ 实验表明:在计数初值为奇数的前提下,实际装入的初值以及自动重装如初值均为编程时写入的初值减 1。在输出正脉冲期间,计数值减到 -2 时,输出端变负,初值(为

编程时写入的初值减 1) 自动重装。在输出负脉冲期间, 计数值减到 0 时, 输出变正, 内部完成初值重装, 重装的初值也是编程时写入的初值减 1。

⑤ 如果将开关 K 下合使 GATE₀ 为低电平, 则计数器停止计数, 每次按下回车键, 读出的计数值都没有变化。

⑥ 对该实验电路不做任何改动, 仅修改源程序, 改变 8254 的方式控制命令字, 即可测试其他 5 种工作方式的计数过程。

⑦ 将实验电路中的 8254 芯片改换成 8253 计数器, 就可以测试 8253 的 6 种工作方式, 欲测试 8253 方式 3 的计数过程, 仅需对上述源程序稍做修改即可。

8253 不能锁存计数器的状态信息, 因此只需要把源程序中的“读状态字→BL”和“显示状态信息”这 3 条指令删除即可。或者既删除这 3 条指令, 也把锁存命令改为 06H。两种方法都可以。

8253 方式 3 的计数过程也如图 11-15 所示。从中可以看出: 就方式 3 而言, 8253 和 8254 芯片的计数规律是不同的, 但是输出的波形相同。

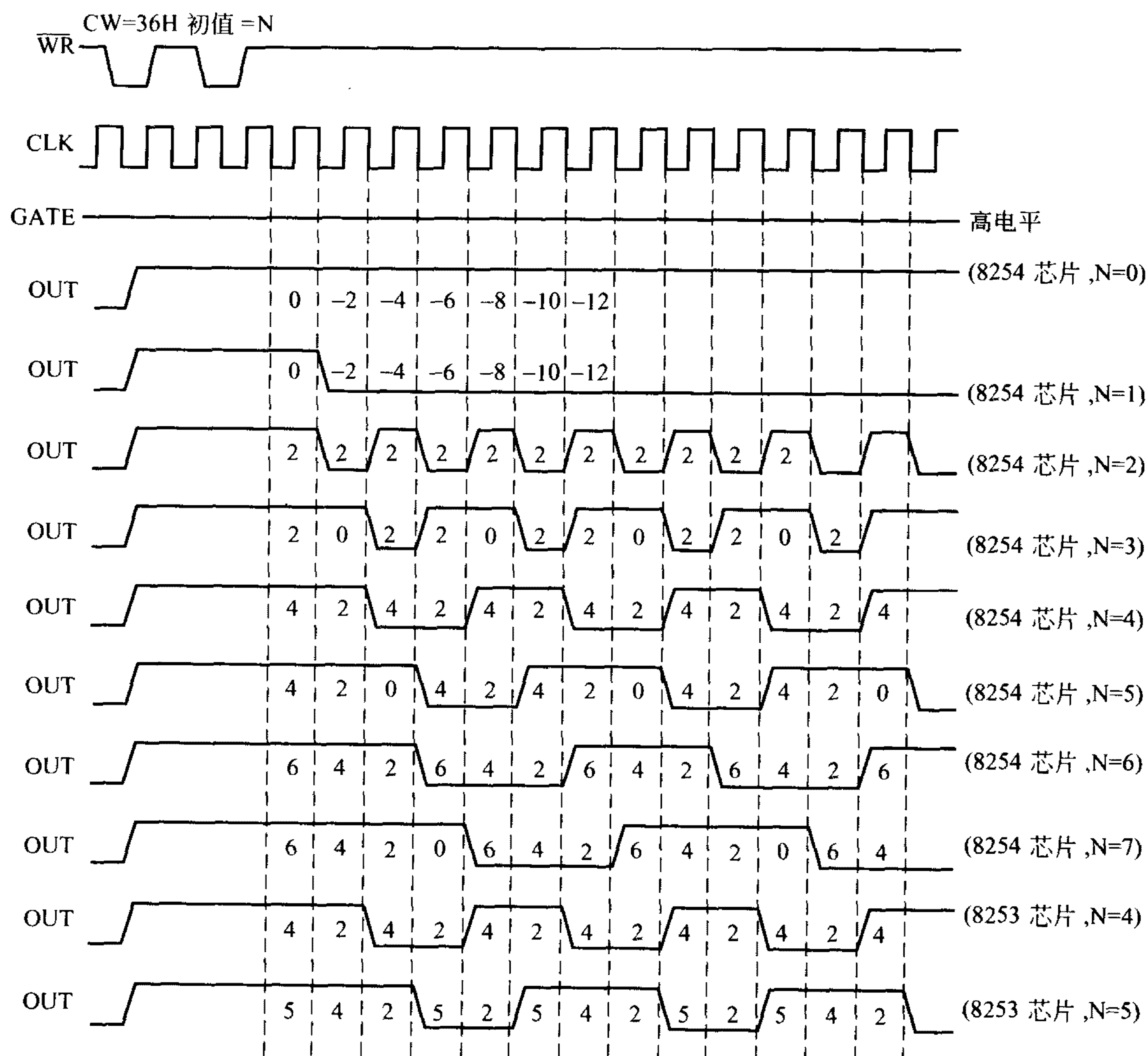


图 11-15 8254/8253 方式 3 计数过程

⑧ 编者对 8254、8254-2、82C54 三种芯片在相同的软硬件环境下进行测试,结果是相同的。

11.3 打印机并行接口

PC 系列机打印设备主要有针式打印机,激光打印机,喷墨打印机等。针式打印机采用点阵式结构由打印头上的打印针在纸上打印小点,构成字符和图形;喷墨打印机是把墨粒子直接喷到打印纸上产生字符和图形;激光打印机利用激光感光技术打印出字符和图形。要说明的是,凡送打印机打印的字符,需用 ASCII 码表示。

本节介绍打印机并行接口。

11.3.1 打印机并行接口标准

打印机并行接口通常采用 Centronics 并行接口标准,共有 36 个引脚信号。表 11-2 给出了 Centronics 标准接口信号的说明。

表 11-2 Centronics 并行打印接口标准

引 脚	信 号 名 称	方 向	功 能 说 明
1	$\overline{\text{STROBE}}$	主机→打印机	数据选通脉冲(低电平接收数据)
2~9	$D_0 \sim D_7$	主机→打印机	8 根数据线
10	$\overline{\text{ACKNLG}}$	主机←打印机	打印机应答信号,表示已收到数据
11	BUSY	主机←打印机	打印机忙,不能接收新的数据
12	PE	主机←打印机	缺纸
13	SLCT	主机←打印机	表示打印机能工作
14	$\overline{\text{AUTO FEEDXT}}$	主机→打印机	打印一行后,自动走纸
15			未用
16	逻辑地		
17	机壳地		
18			未用
19~30	GND		地
31	$\overline{\text{INIT}}$	主机→打印机	初始化打印机
32	$\overline{\text{ERROR}}$	主机←打印机	无纸、脱机、出错指示
33	地		
34			未用
35	电源		+5V
36	$\overline{\text{SLCT IN}}$	主机→打印机	允许打印机工作

在系统连接时,打印机一端是 36 芯的 D 型插座,主机一端是 25 芯的 D 型插座,表 11-3列出了主机和打印机之间打印电缆的连线表。

表 11-3 主机和打印机接口信号连线表

信 号	PC 机并行口 25 芯 D 型插 座引脚	信 号 方 向	打印机并行 口 36 芯 D 型插座引脚	功 能 说 明
D ₀ ~ D ₇	2 ~ 7	主机→打印机	2 ~ 9	数据线(低电平接收数据)
$\overline{\text{STROBE}}$	1	主机→打印机	1	数据选通脉冲
$\overline{\text{ACKNLG}}$	10	主机←打印机	10	打印机应答信号,表示已收到数据
BUSY	11	主机←打印机	11	打印机忙,不能接收新的数据
PE	12	主机←打印机	12	缺纸
SLCT	13	主机←打印机	13	表示打印机能工作
$\overline{\text{AUTO FEEDXT}}$	14	主机→打印机	14	打印一行后,自动走纸
$\overline{\text{ERROR}}$	15	主机←打印机	32	无纸、脱机、出错指示
$\overline{\text{INIT}}$	16	主机→打印机	31	初始化打印机
$\overline{\text{SLCT IN}}$	17	主机→打印机	36	允许打印机工作
GND	18 ~ 25		19 ~ 30、33	地

主机和打印机通信时的联络信号主要是 2 根握手联络信号线 $\overline{\text{STROBE}}$, $\overline{\text{ACKNLG}}$ 和 1 根忙标志线 BUSY,打印字符的 ASCII 码通过 8 根并行数据线传送给打印机。打印过程中数据传送时序如图 11-16 所示。

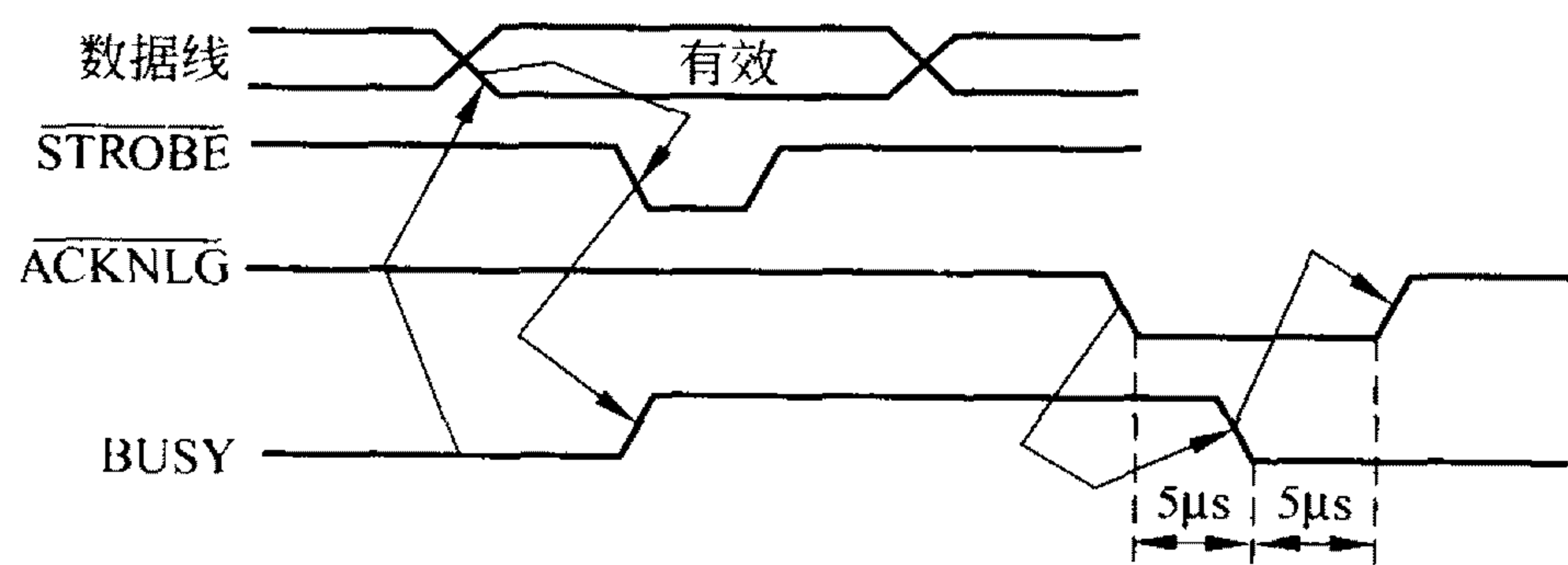


图 11-16 Centronics 并行接口时序

当主机要求打印机工作时,首先测试 BUSY 信号是否为低电平,若 BUSY 为低电平,表示打印机空闲,主机可向打印机输出数据。主机发送一个数据,并且产生 $\overline{\text{STROBE}}$ 选通信号,打印机在 $\overline{\text{STROBE}}$ 下降沿读取数据。打印机接收到数据后,置 BUSY 为高电平,表示打印机不空,不再接收新的数据。打印机接收完数据,发出 $\overline{\text{ACKNLG}}$ 应答信号,通知主机当前数据已取走。在 $\overline{\text{ACKNLG}}$ 变低后 5μs,打印机置 BUSY 为低电平,再过 5μs, $\overline{\text{ACKNLG}}$ 变为高电平,允许 CPU 发送数据。

11.3.2 打印机适配器

PC 系列机的打印机适配器是一种专用的并行接口电路,具有多种形式,但基本功能都相同,均支持各种类型的打印机和主机连接。

PC 系列机一般可配置 3 个打印机适配器 LPT1,LPT2 和单显/打印机,每个打印机并行口占用 3 个 I/O 端口地址,分别对应数据端口,控制端口和状态端口,其端口地址如表 11-4 所示。

表 11-4 打印机 I/O 端口地址分配

打印机接口	数据端口	状态端口	控制端口
LPT1	378H	379H	37AH
LPT2	278H	279H	27AH
单显/打印机	3BCH	3BDH	3BEH

1. 控制寄存器(37AH/27AH)

控制寄存器的功能包括初始化打印机接口,设置中断方式等。其格式如图 11-17 所示。

D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
×	×	×	允许中断	联机	初始化	自动换行	数据选通

图 11-17 控制寄存器格式

D₀: 当 CPU 向打印机发送数据时,用于控制产生数据选通信号。CPU 每输出一个打印字符,该位先置 1,然后再置 0,进行数据选通。

D₁ = 1,控制打印机自动换行。

D₂: 初始化打印机接口和初始化打印机。初始化打印机接口主要设置控制寄存器,包括设置中断方式,自动换行方式,联机等;初始化打印机在打印机加电时自动完成,也可通过将 D₂ 先清零,然后延迟 50μs,再将 D₂ 置 1,这样,INIT上产生一个 50μs 的负脉冲,清除打印机缓冲区。

D₃ = 1,主机与打印机处于联机状态。

D₄ = 1,允许打印机适配器提出中断请求。即当ACKNLG为低电平时,打印机适配器向 8259A 发中断请求信号 IRQ₇。

D₅ ~ D₇,未用。

2. 状态寄存器(379H/279H)

状态寄存器提供打印机工作状态,供 CPU 读取。

状态寄存器格式如图 11-18 所示。D₆ ~ D₃ 位的状态电平来自 25 芯插座同名信号线,D₇ 位是忙标志线的反相电平。

D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
BUSY	ACKNLG	PE	SLCT	ERROR	×	×	×
忙/闲	应答	缺纸	联机	错误标志			

图 11-18 状态寄存器格式

D₀ ~ D₂,未用。

D₃ = 0,表示打印机出错,包括脱机,缺纸及其他错误。

$D_4 = 1$, 表示主机与打印机处于联机状态。

$D_5 = 1$, 表示打印机缺纸。

$D_6 = 0$, 表示打印机准备就绪, 由打印机发 $\overline{\text{ACKNLG}}$ 信号, 使 $D_6 = 0$ 。

$D_7 = 0$, 表示打印机忙, 不能接收主机发送的数据。

11.3.3 打印机接口编程

主机对打印机的控制可以通过对其适配器的编程来实现, 也可通过 BIOS 或 DOS 功能调用来实现。凡送打印机的字符, 全部需要用 ASCII 码表示。

1. 对打印机适配器端口直接编程

对打印机适配器端口直接编程可实现查询方式或中断方式的字符打印。

(1) 查询方式

查询方式首先不断测试 BUSY 信号, 如果 BUSY 信号为低电平, 打印机空闲, 则发送欲打印字符信息, 同时发送选通信号 $\overline{\text{STROBE}}$, 将字符信息送入打印机数据缓冲区。

【例 11.3.1】 查询方式打印程序。

通过系统并行口 1, 打印一行字符“HELLO!”。并行口 1 的数据端口地址为 378H, 控制端口地址为 37AH, 状态端口地址为 379H。

【程序清单】

```

;FILENAME: 1131. ASM
DATA    SEGMENT
BUFFER  DB          'HELLO !',0DH,0AH
COUNT EQU          $-BUFFER
DATA    ENDS
CODE    SEGMENT
        ASSUME      CS: CODE,DS: DATA
BEG:    MOV         AX,DATA
        MOV         DS,AX
        MOV         BX,OFFSET BUFFER
        MOV         CX,COUNT
CHECK:  MOV         DX,379H
        IN          AL,DX                ;状态字→AL
        AND         AL,80H
        JZ          CHECK                ;忙,转移
        MOV         AL,[BX]
        MOV         DX,378H
        OUT         DX,AL                ;输出一个字符编码
        MOV         AL,00001101B
        MOV         DX,37AH
        OUT         DX,AL
        MOV         AL,00001100B

```

```

                OUT        DX,AL                ;发选通脉冲
                INC        BX
                LOOP       CHECK                ;循环计数
                MOV        AH,4CH
                INT        21H
CODE           ENDS
                END        BEG

```

(2) 中断方式

中断方式打印程序的设计要点是：

- ① 首先要设置打印适配器中的控制寄存器,关键是令 D_4 位=1,允许打印机中断。
- ② 系统机并口 1 的中断类型为“0FH”,程序要预先置换 0FH 型中断向量,把打印机中断服务程序的入口地址写入 $4 \times 0FH \sim 4 \times 0FH + 3$ 单元中,还要将主 8259 中断屏蔽寄存器 D_7 位置 0,开放打印机中断。
- ③ 做完准备工作后,程序应向打印机输出一个字符启动打印机中断(本例使用回车键)。打印机收到字符后,通过 \overline{ACKNLG} 联络线,向打印适配器回送一个负脉冲,适配器将其反向送往主 8259 的 IR_7 ,作为打印适配器的中断请求,CPU 响应中断后,转入打印机中断服务程序,继续发送下一个字符。

【例 11.3.2】 用中断方式打印 4 行“HELLO”。

并行口 1 的数据端口地址为 378H,控制端口地址为 37AH,状态端口地址为 379H。

【程序清单】

```

;FILENAME: 1132. ASM
DISP      MACRO      VAR
            MOV        AH,9
            MOV        DX,OFFSET VAR
            INT        21H
            ENDM
PRINT     MACRO                ;打印一个字符
            MOV        DX,378H
            MOV        AL,[BX]
            OUT        DX,AL    ;输出一个字符编码
            MOV        DX,37AH
            MOV        AL,00011101B
            OUT        DX,AL
            MOV        AL,00011100B
            OUT        DX,AL    ;发选通脉冲
            INC        BX      ;地址加一
            ENDM
SCAN      MACRO                ;测试打印机状态
            MOV        DX,379H
            IN         AL,DX    ;状态字→AL
            TEST       AL,20H

```

```

JNZ ERR_1 ; 缺纸, 转移
TEST AL, 10H
JZ ERR_2 ; 未联机, 转移
TEST AL, 08H
JZ ERR_3 ; 打印机出错, 转移
ENDM

DATA SEGMENT
OLD0F DD ?
BUF DB 0DH, 4 DUP('HELLO !', 0DH, 0AH), 0
MSG1 DB 'Paper is not ready ! $ '
MSG2 DB 'Printer is not ready ! $ '
MSG3 DB 'Printer error! $ '
MSG4 DB 'End ! $ '
DATA ENDS
CODE SEGMENT
ASSUME CS: CODE, DS: DATA
BEG: MOV AX, DATA
MOV DS, AX
SCAN ; 测试打印机状态
CLI
CALL READ0F ; 转移 0FH 型中断向量
CALL WRITE0F ; 写入打印机中断向量
CALL I8259 ; 开放打印机中断
STI
MOV BX, OFFSET BUF
PRINT ; 启动打印机中断
CHECK: CMP BYTE PTR [BX], 0
JZ SUCCESS ; 打印结束, 转移
SCAN ; 测试打印机状态
JMP CHECK
ERR_1: DISP MSG1 ; 显示"缺纸"
JMP EXIT
ERR_2: DISP MSG2 ; 显示"打印机未准备好"
JMP EXIT
ERR_3: DISP MSG3 ; 显示"打印机出错"
JMP EXIT
SUCCESS: DISP MSG4 ; 显示结束信息
CALL RESET ; 恢复系统资源
EXIT: MOV AH, 4CH
INT 21H

;-----
SERVICE PROC
PUSH AX ; 保护现场
PUSH DX ; 保护现场

```



```

                PRINT                                ; 打印一个字符
                MOV     AL,20H                        ; 中断结束命令
                OUT     20H,AL                        ; → 主 8259
                POP     DX                            ; 恢复现场
                POP     AX                            ; 恢复现场
                IRET                                     ; 中断返回
SERVICE       ENDP
;-----
READ0F         PROC                                ; 转移 0FH 型中断向量
                MOV     AX,350FH
                INT     21H
                MOV     WORD PTR OLD0F,BX
                MOV     WORD PTR OLD0F+2,ES
                RET
READ0F         ENDP
;-----
WRITE0F        PROC                                ; 写入打印机中断向量
                PUSH    DS
                MOV     AX,CODE
                MOV     DS,AX
                MOV     DX,OFFSET SERVICE
                MOV     AX,250FH
                INT     21H
                POP     DS
                RET
WRITE0F        ENDP
;-----
I8259          PROC                                ; 开放打印机中断
                IN      AL,21H
                AND     AL,01111111B
                OUT     21H,AL
                RET
I8259          ENDP
;-----
RESET          PROC                                ; 恢复系统资源
                IN      AL,21H
                OR      AL,10000000B
                OUT     21H,AL                        ; 屏蔽打印机中断
                MOV     DX,WORD PTR OLD0F
                MOV     DS,WORD PTR OLD0F+2
                MOV     AX,250FH
                INT     21H                            ; 恢复 0FH 型中断向量
                RET
RESET          ENDP

```

```
CODE      ENDS
          END          BEG
```

2. BIOS 功能调用

在 BIOS 系统中提供了打印机管理程序,用户可使用 INT 17H 功能调用,完成字符打印。

【INT 17H 0 号子功能】 打印一个字符。

入口参数: AL=打印字符的 ASCII 码。

DX=打印机号(0~2)。

出口参数: AH=打印机状态。

【INT 17H 1 号子功能】 初始化打印机。

入口参数: DX=打印机号(0~2)。

出口参数: AH=打印机状态。

【INT 17H 2 号子功能】 读打印机状态。

入口参数: DX=打印机号(0~2)。

出口参数: AH=打印机状态。

注意: 并口 1 的打印机号为 0,返回到 AH 的打印机状态格式如图 11-19 所示。

D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
BUSY	ACKNLG	PE	SLCT	ERROR	×	×	TIMEOUT
忙/闲	应答	缺纸	联机	错误标志			超时

图 11-19 状态寄存器格式(INT 17H,2 号子功能出口参数)

D₀=1,表示超时。当打印机处于忙状态超过 1 秒,表明打印机出现意外故障,CPU 测试到超时状态,可以从循环查询中退出。

D₁、D₂,未用。

D₃=1,表示出错。该位功能和打印机适配器状态寄存器 D₃ 位一致,但信号极性相反。

D₄=1,处于联机状态。该位功能和打印机适配器状态寄存器 D₄ 位一致。

D₅=1,表示缺纸。该位功能和打印机适配器状态寄存器 D₅ 位一致。

D₆=1,应答信号有效。该位功能和打印机适配器状态寄存器 D₆ 位一致,但信号极性相反。

D₇=1,表示打印机空闲。该位功能和打印机适配器状态寄存器 D₇ 位一致。

3. DOS 功能调用

用户可调用 INT 21H 的 5 号子功能,完成字符打印功能。

【INT 21H 5 号子功能】 打印一个字符。

入口参数: DL=打印字符的 ASCII 码。

出口参数：无。

习 题

1. 并行接口芯片有什么特点？一般应用于什么场合？
2. 8255A 各端口有几种工作方式？
3. 8255A 的 3 个端口在使用时有何区别？
4. 8255A 工作在方式 1 和方式 2 时，哪些引脚是联络线？这些联络信号有效时代表什么物理意义？
5. 当 CPU 用查询方式和 8255A 交换信息时，应查询哪些信号？当 CPU 用中断方式和 8255A 交换信息时，利用哪些端子提中断请求？
6. 8255A 的方式选择控制字和 C 端口置 0/置 1 控制字都是写入控制端口的，8255A 是怎样识别的？
7. 8255A 工作在方式 1 输入时，如果 CPU 用查询方式和 8255A 交换信息，为什么不查询 $\overline{\text{STB}}$ 信号？
8. 8255A 工作在方式 1 或者方式 2 时，设置中断允许，应采取什么措施？
9. 说明打印机 Centronics 并行接口时序。
10. 简述采用查询方式对打印机接口编程的工作过程。

DMA 控制器

12.1 概 述

直接存储器存取(Direct Memory Access,DMA)方式,是用硬件实现存储器和存储器之间或存储器和 I/O 设备之间直接进行的高速数据传送,不需要 CPU 的干预。这种方式通常用来传输数据块。

实现 DMA 传送的关键器件是 DMA 控制器,称为 DMAC。

DMA 传送包括:

- ① DMA 读传送: RAM→I/O 端口。
- ② DMA 写传送: I/O 端口→RAM。
- ③ 存储单元传送: RAM→RAM。

DMA 传送过程如图 12-1 所示。

从图中可见:系统总线受到 CPU 和 DMAC 二个器件的控制,即 CPU 可以向地址总线、数据总线和控制总线发送信息,DMAC 也可以向地址总线、数据总线和控制总线发送信息。但是在同一时间,系统总线只能受一个器件控制。CPU 控制总线时,DMAC 必须与总线脱离;而 DMAC 控制总线时,CPU 必须与总线脱离。因此,CPU 与 DMAC 之间必须有“联络信号”。

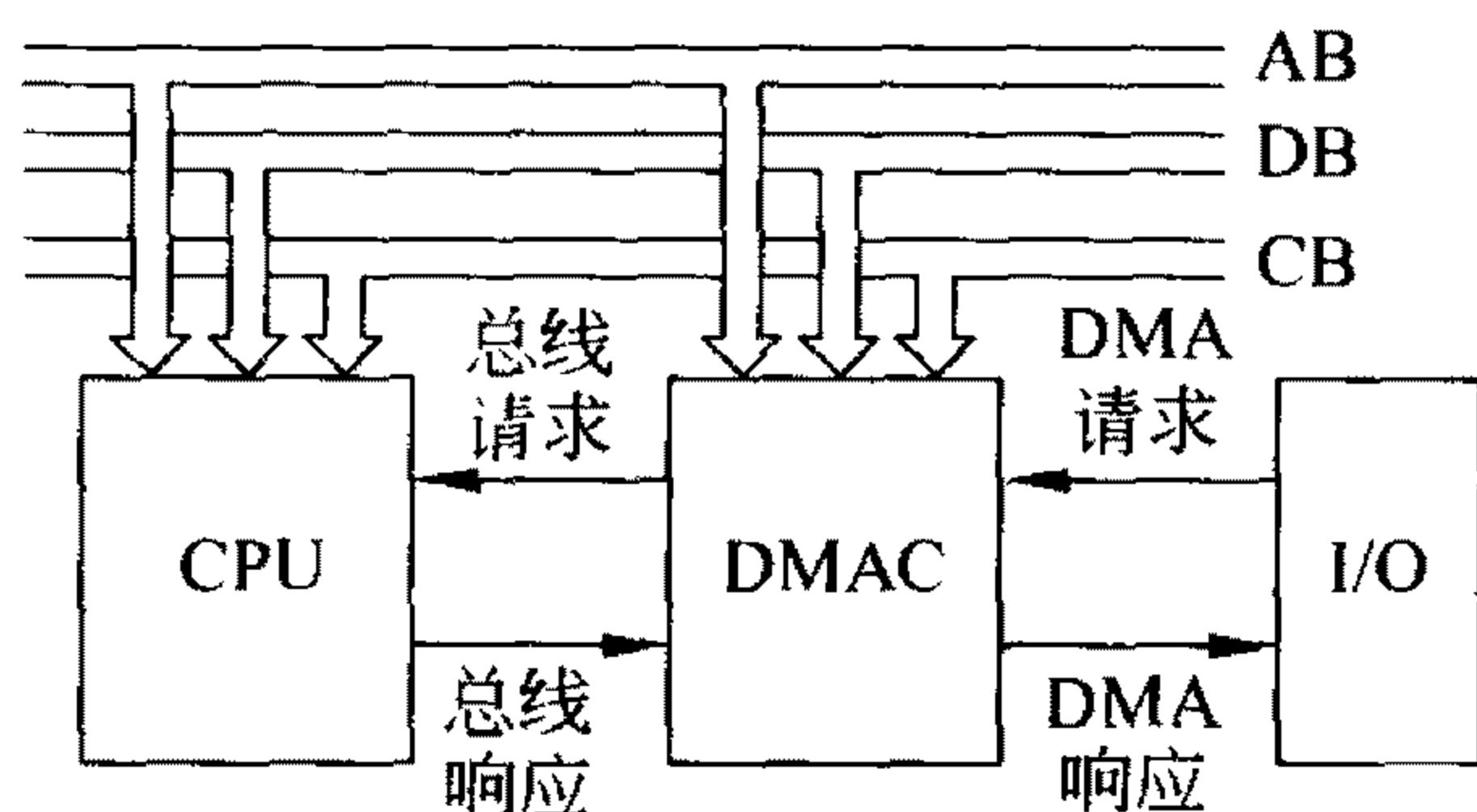


图 12-1 DMA 传送过程示意图

DMA 传送的工作过程如下:

- ① I/O 端口向 DMA 控制器发出 DMA 请求,请求传送数据。
- ② DMA 控制器在接到 I/O 端口的 DMA 请求后,向 CPU 发出总线请求信号,请求 CPU 脱离系统总线。
- ③ CPU 在执行完当前指令的当前总线周期后,向 DMA 控制器发出总线响应信号。
- ④ CPU 随即和系统的控制总线、地址总线及数据总线脱离关系,处于等待状态,由 DMA 控制器接管三总线控制权。
- ⑤ DMA 控制器向 I/O 端口发出 DMA 应答信号。
- ⑥ DMA 控制器把进行 DMA 传送涉及到的 RAM 地址→地址总线;如果进行 I/O 端口→RAM 传送,DMAC 向 I/O 端口发出 I/O 读命令,向 RAM 发出存储器写命令;如果进行 RAM→I/O 端口传送,DMAC 向 RAM 发出存储器读命令,向 I/O 端口发出 I/O

写命令,从而完成一个字节的传送。

⑦ 当设定的字节数传送完毕后,DMA 传输过程结束,也可以由来自外部的终止信号迫使传输过程结束。DMA 传送结束后,DMA 控制器就将总线请求信号变为无效,并放弃对总线的控制,CPU 检测到总线请求信号无效后,也将总线响应信号变为无效,于是,CPU 重新控制三总线,继续执行被中断的当前指令的其他总线周期。

从以上分析可以看出,DMA 传送比中断方式更快。它们的特点比较如下:

① DMA 传送比中断传送的速度快。DMA 传送一个字节只占用 CPU 的一个总线周期,而中断传送方式是由 CPU 通过程序来实现的,每次执行中断服务程序,CPU 要保护断点,在中断服务程序中,需要保护现场和恢复现场,需要执行若干条指令才能传送一个字节。

② DMA 响应比中断响应的速度快。中断方式是在 CPU 的当前指令(一条指令需要执行若干个总线周期)执行完才能响应中断请求,而 DMA 方式是在 CPU 当前指令的一个总线周期执行完就响应 DMA 请求。

③ 中断请求分为外部中断(由外部硬件产生的)和内部中断(由执行指令产生的),DMA 请求也有两种方式:可以由硬件发出,也可以由软件发出。

12.2 8237A DMA 控制器

8237A 是微型计算机系统中实现 DMA 功能的大规模集成电路控制器,PC/AT 使用两片 8237A,在高档微型计算机中常使用多功能芯片取代 8237A,但多功能芯片中的 DMA 控制器与 8237A 的功能基本相同。

12.2.1 8237A 的内部结构和引脚功能

8237A 是具有 4 个独立 DMA 通道的可编程 DMA 控制器(DMAC),它使用单一 +5V 电源,单相时钟,40 引脚双列直插式封装。在实际应用中,8237A 必须与一片 8 位锁存器一起使用,才能形成一个完整的 4 通道 DMA 控制器。8237A 初始化后,可以控制每一个通道在存储器和 I/O 端口之间以最高 1.6 兆波特的速率传送最多达 64KB 的数据块,而不需要 CPU 介入。

1. 8237A 的基本功能

8237A 的基本功能如下:

- ① 在一个芯片中有 4 个独立的 DMA 通道。
- ② 每一个通道的 DMA 请求都可以被允许或禁止。
- ③ 每一个通道的 DMA 请求有不同的优先级,可以是固定优先级,也可以是循环优先级。
- ④ 每一个通道一次传送的最大字节数为 64KB。
- ⑤ 8237A 提供 4 种传送方式,它们是:单字节传送方式、数据块传送方式、请求传送

方式和级连传送方式。

2. 8237A 的内部结构

8237A 的内部结构如图 12-2 所示。

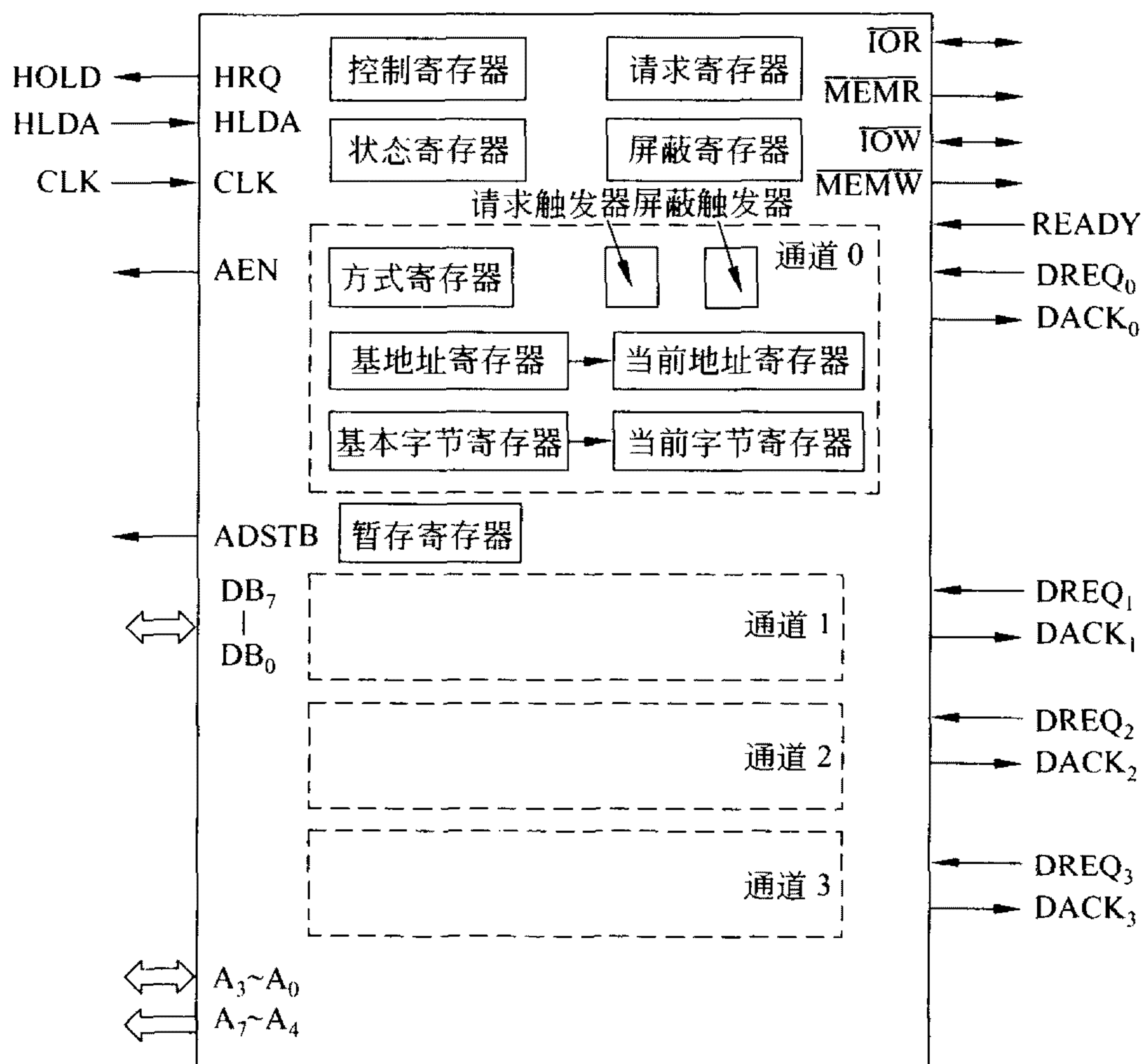


图 12-2 8237A 的内部结构框图

(1) DMA 通道

8237A 内部包含 4 个独立通道,每个通道包含两个 16 位的地址寄存器、两个 16 位的字节寄存器、一个 6 位的方式寄存器、一个 DMA 请求触发器和一个 DMA 屏蔽触发器,此外,4 个通道共用一个 8 位控制寄存器,一个 8 位状态寄存器,一个 8 位暂存器,一个 8 位屏蔽寄存器和一个 8 位请求寄存器。

(2) 读/写逻辑

当 CPU 对 8237A 初始化或对 8237A 寄存器进行读操作时,8237A 就像 I/O 端口一样被操作,读/写逻辑接收 IOR 或 IOW 信号,当 IOR 为低电平时,CPU 可以读取 8237A 的内部的寄存器值,当 IOW 为低电平时,CPU 可以将数据写入 8237A 的内部寄存器中。

在 DMA 传送期间,系统由 8237A 控制总线,此时,8237A 分两次向地址总线上送出要访问的内存单元 20 位物理地址中的低 16 位,8237A 输出必要的读/写信号,这些信号为 I/O 读信号 IOR、I/O 写信号 IOW、存储器读信号 MEMR、存储器写信号 MEMW。

(3) 控制逻辑

在 DMA 周期内,控制逻辑通过产生相应的控制信号和 16 位要存取的内存单元地址来控制 DMA 的操作步骤。初始化时,通过对方式寄存器编程,使控制逻辑可以对各个通

道的操作进行控制。

3. 8237A 引脚功能

图 12-3 为 8237A 引脚图。

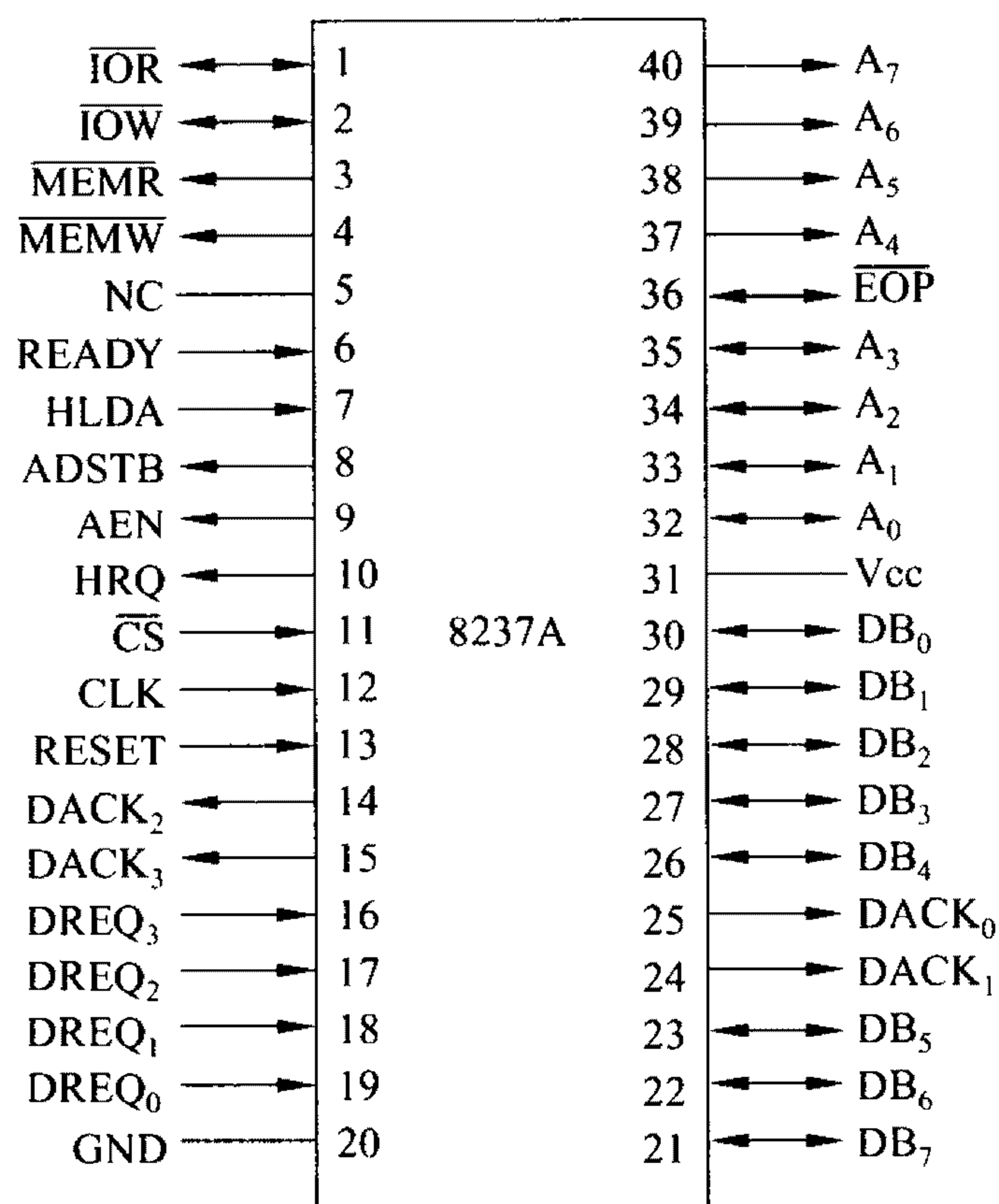


图 12-3 8237A 引脚图

CLK: 时钟输入端, 接到 8284 时钟发生器的输出引脚, 用来控制 8237A 内部操作定时和 DMA 传送时的数据传送速率。8237A 的时钟频率为 3MHz, 8237A-5 的时钟频率为 5MHz, 8237A-5 是 8237A 的改进型, 工作原理及使用方法和 8237A 相同。

$\overline{\text{CS}}$: 片选输入端, 低电平有效。

RESET: 复位输入端, 高电平有效。RESET 有效, 则屏蔽寄存器被置 1 (4 个通道均禁止 DMA 请求), 其他寄存器被清零, 8237A 处于空闲周期, 所有控制线都处于高阻状态, 禁止 4 个通道的 DMA 操作。复位后 8237A 要进入 DMA 操作必须重新初始化。

READY: “准备就绪”信号输入端, 高电平有效。当所选择的存储器或 I/O 端口的速度较慢, 需要延长传输时间, 可使 READY 处于低电平, 8237A 会自动在存储器读和存储器写周期中插入等待周期。若 READY 变为高电平, 表示存储器或 I/O 设备准备就绪。

ADSTB: 地址选通输出信号, 高电平有效, 该信号有效时, 8237A 当前地址寄存器的高 8 位经数据总线 $\text{DB}_7 \sim \text{DB}_0$ 锁存到外部地址锁存器中。

AEN: 地址允许输出信号, 高电平有效, AEN 把外部地址锁存器中锁存的高 8 位地址输出到地址总线上, 与芯片直接输出的低 8 位地址共同构成内存单元的 16 位偏移地址。

$\overline{\text{MEMR}}$: 存储器读信号, 低电平有效。输出信号, 只用于 DMA 传送。在 DMA 读周

期期间,从所寻址的存储器单元中读出数据。

$\overline{\text{MEMW}}$: 存储器写信号,低电平有效。输出信号,只用于 DMA 传送。在 DMA 写周期期间,将数据写入所寻址的存储单元。

$\overline{\text{IOR}}$: I/O 读信号,低电平有效,双向。CPU 控制总线时,是输入信号,CPU 读 8237A 内部寄存器的内容;当 8237A 控制总线时,是输出信号,与 $\overline{\text{MEMW}}$ 配合,控制数据由 I/O 端口传送到存储器。

$\overline{\text{IOW}}$: I/O 写信号,低电平有效,双向。CPU 控制总线时,是输入信号,CPU 将内容写入 8237A 内部寄存器(初始化);当 8237A 控制总线时,是输出信号,与 $\overline{\text{MEMR}}$ 配合,把数据从存储器传送到 I/O 端口。

$\overline{\text{EOP}}$: DMA 传送结束信号,低电平有效,双向。当 DMA 控制的任一通道计数结束时, $\overline{\text{EOP}}$ 输出低电平,表示 DMA 传送结束;当外部向 DMA 控制器输入 $\overline{\text{EOP}}$ 信号时,DMA 传送过程将被强迫结束。无论是从外部终止 DMA 过程,还是内部计数结束引起 DMA 过程结束,都会使 DMA 控制器的内部寄存器复位。

$\text{DREQ}_0 \sim \text{DREQ}_3$: DMA 请求输入信号,编程设定有效电平,是外设为获得 DMA 服务而送到各个通道的请求信号。固定优先级时, DREQ_0 优先级最高, DREQ_3 优先级最低;循环优先级时,某通道的 DMA 请求被响应后,便降为最低级。8237A 用 DACK 信号作为对 DREQ 的响应,因此在相应的 DACK 信号有效之前, DREQ 信号必须保持有效。

$\text{DACK}_0 \sim \text{DACK}_3$: DMA 对各个通道请求的响应信号,编程设定输出的有效电平。8237A 接收到通道请求,向 CPU 发出 DMA 请求信号 HRQ ,当 8237A 获得 CPU 送来的总线允许信号 HLDA 后,便产生 DACK 信号送到相应的 I/O 端口,表示 DMA 控制器响应外设的 DMA 请求,进入 DMA 传送过程。

HRQ : 8237A 输出到 CPU 的总线请求信号,高电平有效。当外设的 I/O 端口要求 DMA 传送时,向 DMA 控制器发送 DREQ 信号,如果相应通道屏蔽位为 0,即 DMA 请求未被屏蔽,则 DMA 控制器的 HRQ 端输出为有效电平,向 CPU 提出总线请求。

HLDA : 总线响应信号,高电平有效,CPU 对 HRQ 信号的应答信号。CPU 接收到 HRQ 信号后,在当前总线周期结束之后让出总线控制权,并使 HLDA 信号有效。

$A_3 \sim A_0$: 地址总线低 4 位,双向。当 CPU 控制总线时,是地址输入线,CPU 用这 4 条地址线对 DMA 控制器的内部寄存器进行寻址,完成对 DMA 控制器的编程;当 8237A 控制总线时,这 4 条线输出要访问的存储单元的最低 4 位地址。

$A_7 \sim A_4$: 地址线,输出,只用于 DMA 传送时,输出要访问的存储单元低 8 位地址中的高 4 位。

$\text{DB}_7 \sim \text{DB}_0$: 8 位双向数据线,与系统数据总线相连。在 CPU 控制总线时,CPU 可以通过 I/O 读命令从 DMA 控制器中读取内部寄存器的内容送到 $\text{DB}_7 \sim \text{DB}_0$,以了解 8237A 工作情况,也可以通过 I/O 写命令对 DMA 控制器的内部寄存器进行编程。在 DMA 控制器控制总线时, $\text{DB}_7 \sim \text{DB}_0$ 输出要访问的存储单元的高 8 位地址($A_{15} \sim A_8$),通过 ADSTB 锁存到外部地址锁存器中,并和 $A_7 \sim A_0$ 输出的低 8 位地址一起构成 16 位地址。

8237A 仅支持 64KB 寻址,为了访问超出 64KB 范围的其他地址空间,系统中增设了“页面寄存器”,在 PC/XT 微型计算机系统中,每一通道的页面寄存器是 4 位的寄存器。当一个 DMA 操作周期开始时,相应的页面寄存器内容就放到系统地址总线 $A_{19} \sim A_{16}$ 上,和 8237A 送出的 16 位低地址一起,构成 20 位物理地址。

12.2.2 8237A 内部寄存器

8237A 内部寄存器分为 4 个通道共用的寄存器和各个通道专用的寄存器两类。

1. 控制寄存器

8237A 的 4 个通道共用一个控制寄存器。编程时,由 CPU 写入控制字,而由复位信号(RESET)或软件清除命令清除它。控制寄存器格式如图 12-4 所示。

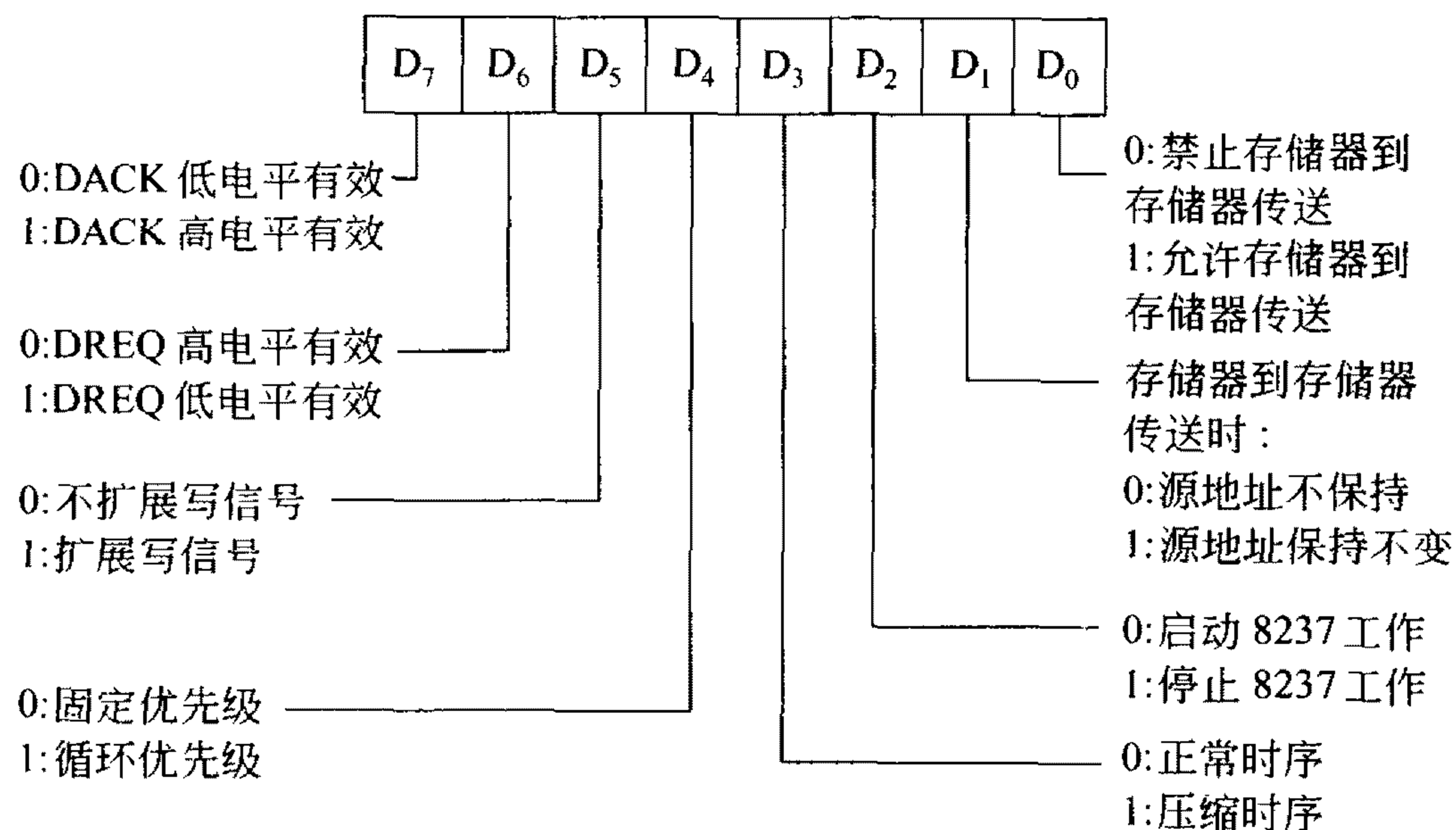


图 12-4 8237A 控制寄存器格式

① D_0 : 规定是否工作在存储器到存储器传送方式。

8237A 约定: 进行存储器之间数据传送时,由通道 0 提供源地址,通道 1 提供目标地址并进行字节计数。每传送一个字节需要两个总线周期,第一个总线周期将源地址单元的数据读入 8237A 的暂存器中,第二个总线周期将暂存器的内容送到数据总线上,随后在写信号的作用下,将数据写入目标地址单元。

② D_1 : 进行存储器到存储器传送时,起控制作用。

③ D_2 : 启动和停止 8237A 的工作。

④ D_3 : 8237A 可以用正常时序或压缩时序工作。如果系统各部分速度较高,要提高 DMA 传输的数据吞吐量,可以采用压缩时序。

⑤ D_4 : 选择各通道 DMA 请求的优先级。当 $D_4 = 0$ 时,为固定优先级,即通道 0 优先级最高,通道 3 最低;当 $D_4 = 1$ 时,为循环优先级,即在每次 DMA 服务之后,各个通道优先级要发生变化,例如,某次传输前的优先级次序为 3-0-1-2,那么在通道 3 进行一次传输之后,优先级次序变为 0-1-2-3,如果这时通道 0 没有 DMA 请求,而通道 1 有 DMA 请求,那么,在通道 1 完成 DMA 传输后,优先级次序变成 2-3-0-1。

DMA 的优先级排序只是用来决定同时请求 DMA 服务通道的响应次序。任何一个通道一旦进入 DMA 服务后,其他通道必须等到该通道服务结束后,才可进行 DMA 服务。

⑥ D_5 : 若 $D_5 = 1$, 选择扩展的写信号 ($\overline{IOW}/\overline{MEMW}$ 比正常时序提前一个状态周期)。

⑦ D_6 、 D_7 : 确定 DREQ 和 DACK 的有效电平极性。对这两位如何设置,取决于 I/O 端口对 DREQ 信号和 DACK 信号的极性要求。

控制字是 4 个通道必须共同遵循的原则。

在 PC 系列机中, BIOS 初始化时, 已将控制寄存器设定为 00H: 禁止存储器到存储器传送, 允许读/写操作, 使用正常时序, 固定优先级, 不扩展写信号, DREQ 高电平有效, DACK 低电平有效, 用户不应当改写。

2. 方式寄存器

8237A 每个通道有一个方式寄存器, 4 个通道的方式寄存器共用一个端口地址, 方式选择命令字的格式, 如图 12-5 所示。方式字的最低两位进行通道选择, 写入命令字之后, 8237A 将根据 D_1 、 D_0 的编码把方式寄存器的 $D_7 \sim D_2$ 位送到相应通道的方式寄存器中, 从而确定该通道的传送方式、数据传送类型。8237A 各通道的方式寄存器是 6 位的, CPU 不可寻址!

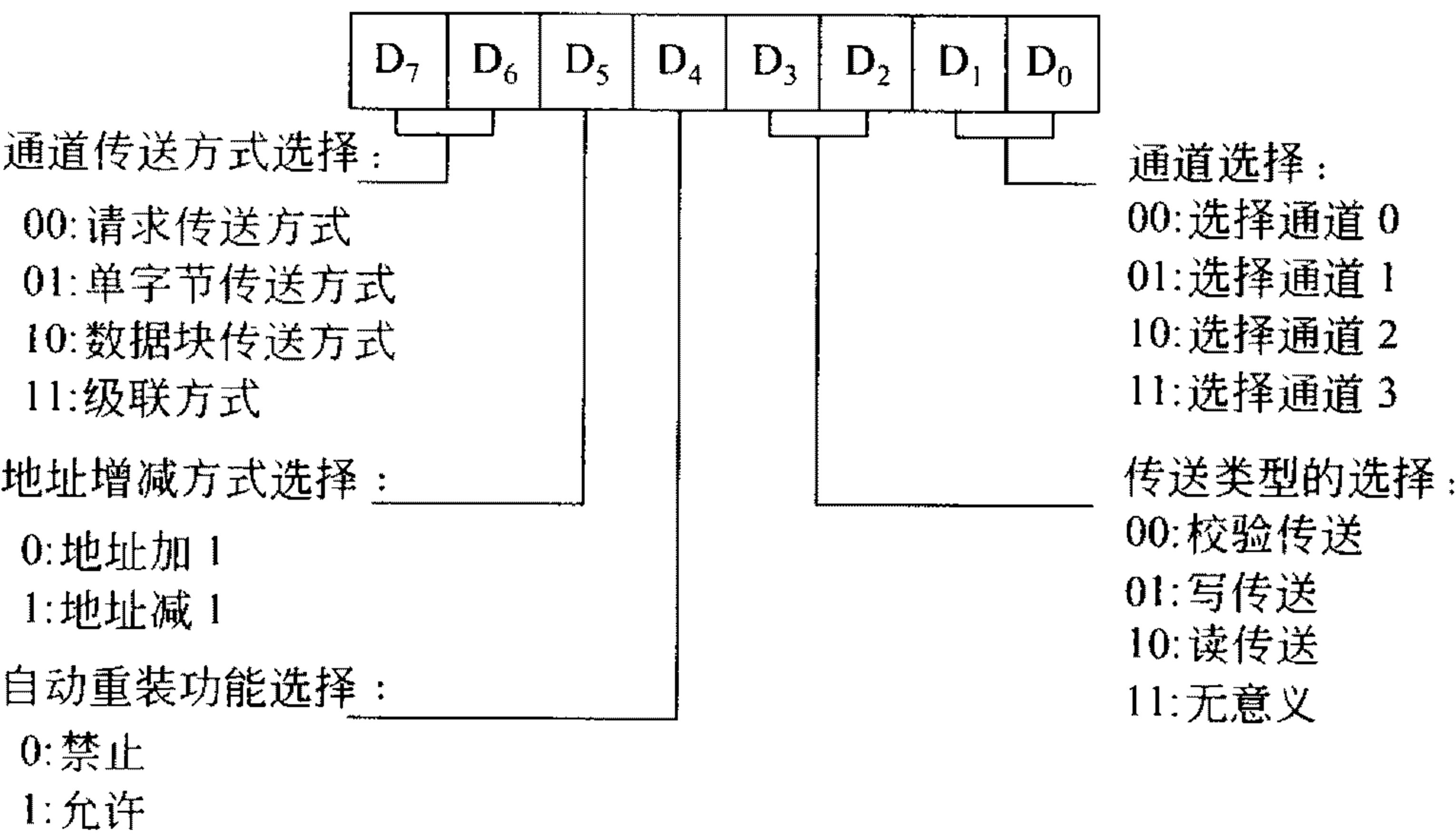


图 12-5 8237A 方式选择命令字格式

8237A 提供 4 种传送方式, 每个通道可以用 4 种方式之一进行工作。方式选择命令字的 D_7 位、 D_6 位确定通道传送方式。

① 单字节传送方式。每次 DMA 操作只传送一个字节的数 据, 然后自动把总线控制权交给 CPU, 让 CPU 占用至少一个总线周期。若有新的 DMA 请求, 8237A 将向 CPU 发出总线请求, 等到获得总线控制权后, 再进行下一个字节数据的传送。

② 数据块传送方式。进入 DMA 操作后, 连续传送数据, 直到整个数据块全部传送完毕。

③ 请求传送方式。该方式与数据块传送方式类似, 只是在每传输一个字节后,

8237A 都将对 DMA 请求信号 DREQ 进行测试,如检测到 DREQ 端变为无效电平,则马上暂停传输,但测试过程仍然进行。当 DREQ 又变为有效电平时,则在原有基础上继续进行传送,直到结束。

④ 级联传送方式。为了实现 DMA 系统扩展,可以进行 8237A 的级联传送。

方式选择命令字的 D_3 位、 D_2 位,确定了数据传送的类型,即写传送、读传送和校验传送。写传送是将数据从 I/O 端口读出写入存储单元。读传送是将数据从存储单元读出写入 I/O 端口。校验传送是一种虚拟传送,8237A 本身并不进行数据传送,而只是像 DMA 读传送或 DMA 写传送一样产生时序,产生地址信号,但存储器或 I/O 端口的读/写控制信号无效。校验传输一般用于器件测试。

方式选择命令字的 D_4 位为 1 时,通道有“自动重装功能”。

方式选择命令字 D_5 位控制“当前地址寄存器”的地址增减方式,规定地址是增量修改还是减量修改。

3. 地址寄存器

每个通道有一个 16 位的“基地址寄存器”和一个 16 位的“当前地址寄存器”。基地址寄存器存放本通道 DMA 传输时所涉及到的存储区首地址或末地址,这个初始值是在初始化编程时写入的,同时也被写入当前地址寄存器。进行 DMA 传送时,由当前地址寄存器向地址总线提供本次 DMA 传送时的内存地址(低 16 位)。当前地址寄存器的值在每次 DMA 传输后自动加 1 或减 1,为传送下一个字节作准备,在整个 DMA 传送期间,基地址寄存器的内容保持不变。当通道初始化选择“自动重装”功能时,一旦全部字节传送完毕,基地址寄存器的内容自动重新装入当前地址寄存器。

4. 字节寄存器

每个通道有一个 16 位的“基本字节寄存器”和一个 16 位的“当前字节寄存器”。基本字节寄存器存放本通道 DMA 传输时字节数的初值,8237A 规定:初值比实际传输的字节数少 1,初值是在初始化编程时写入的,同时初值也被写入当前字节寄存器。在 DMA 传送时,每传送一个字节,当前字节寄存器自动减 1,当初值由 0 减到 FFFFH 时,产生计数结束信号, \overline{EOP} 端子输出有效电平。当通道初始化选择“自动重装”功能时,一旦全部字节传送完毕,基本字节寄存器的内容自动重新装入当前字节寄存器。基本字节寄存器预置初值后将保持不变,也不能被 CPU 读出,而当前字节寄存器中的内容可以随时由 CPU 读出。

5. 状态寄存器

状态寄存器的格式如图 12-6 所示。

状态寄存器高 4 位表示当前 4 个通道是否有 DMA 请求,低 4 位表示 4 个通道的 DMA 传送是否结束,供 CPU 进行查询。

6. 请求寄存器和屏蔽寄存器

请求寄存器和屏蔽寄存器是 4 个通道公用的寄存器,使用时应写入请求命令字和屏

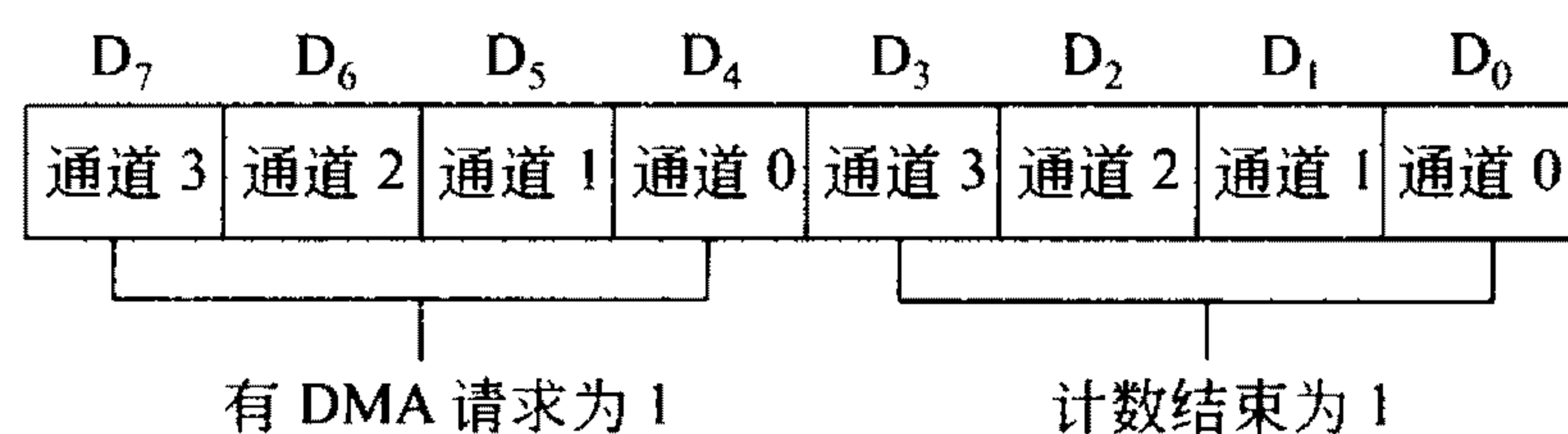


图 12-6 8237A 状态寄存器格式

蔽命令字,其格式如图 12-7 所示。

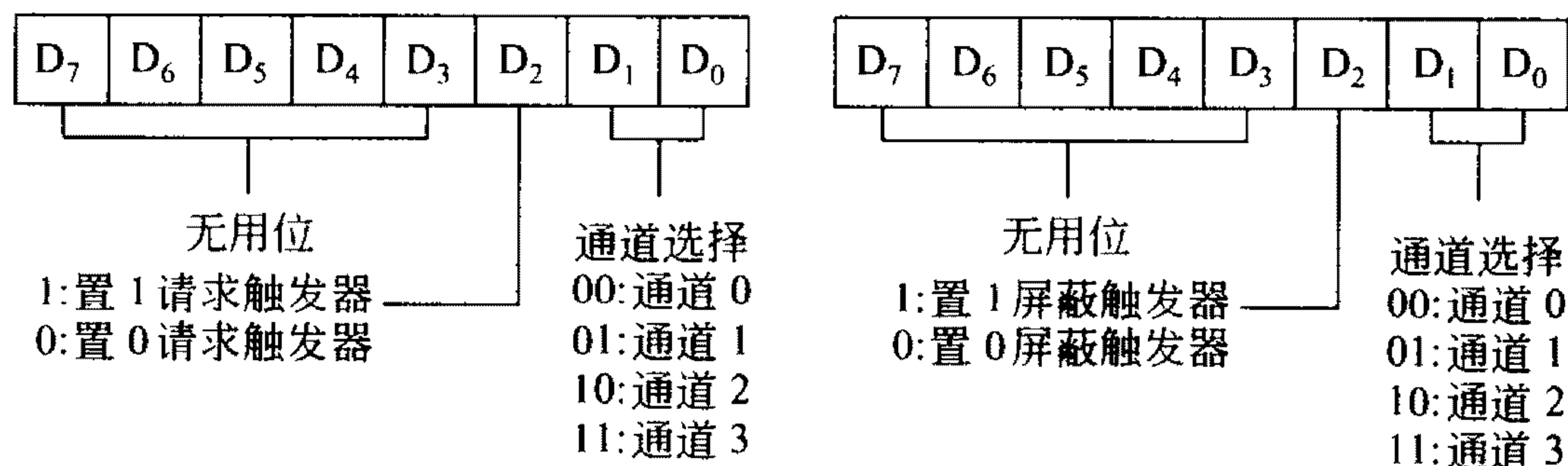


图 12-7 请求寄存器和屏蔽寄存器格式

8237A 根据请求寄存器的 $D_2 \sim D_0$ 位将相应通道的请求触发器置 1(或置 0),使通道提出“软件 DMA 请求”。

8237A 根据屏蔽寄存器的 $D_2 \sim D_0$ 位将相应通道的屏蔽触发器置 1(或置 0),实验表明:当一个通道的屏蔽触发器置 1 后,它将屏蔽来自引脚 DREQ 的硬件 DMA 请求,同时也屏蔽来自请求寄存器的软件 DMA 请求,因此,在对通道初始化之前应使屏蔽触发器置 1,而初始化之后,应使屏蔽触发器置 0。

7. 多通道屏蔽寄存器

8237A 允许使用一个屏蔽字一次完成对 4 个通道的屏蔽设置,格式如图 12-8 所示。其中 $D_0 \sim D_3$ 对应通道 0~通道 3 的屏蔽触发器,某一位为 1,则对应通道的屏蔽触发器置 1。

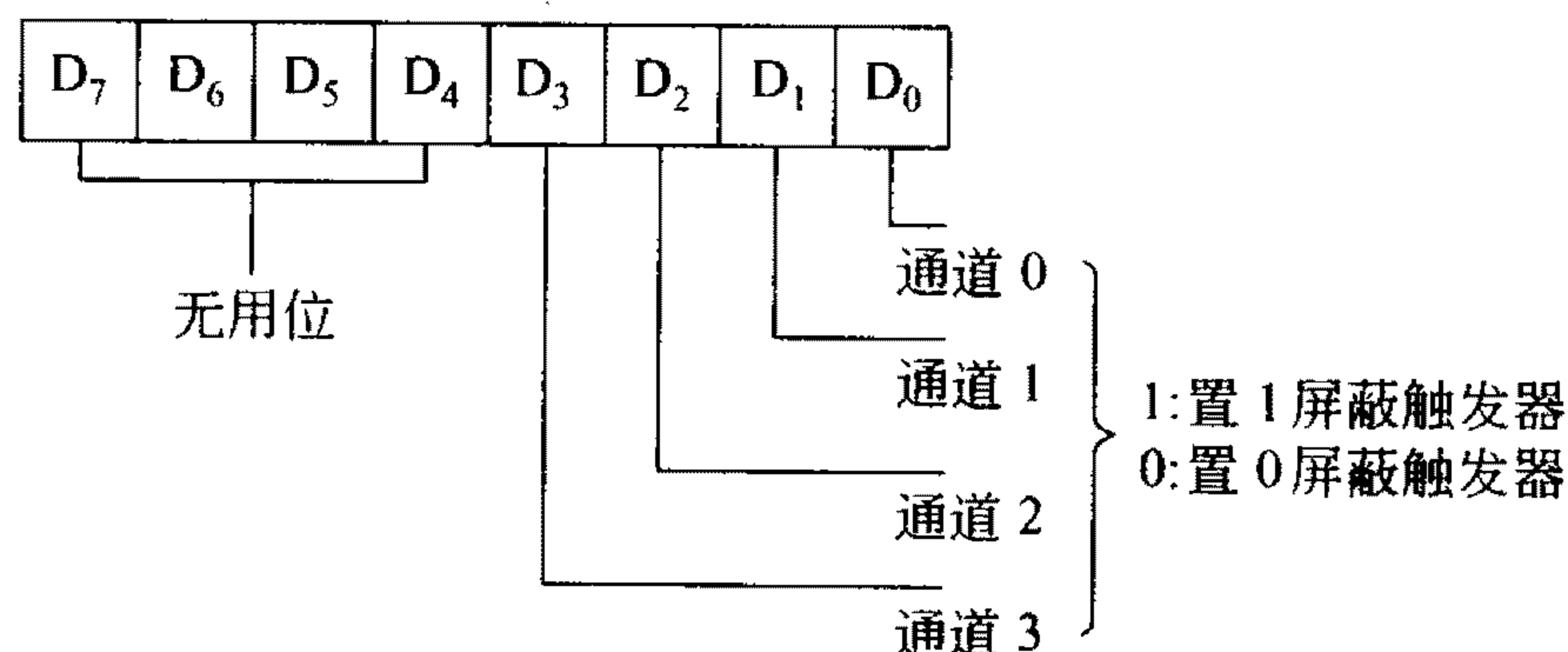


图 12-8 多通道屏蔽寄存器

8. 清屏蔽寄存器

无论 RESET 复位还是软件复位,屏蔽寄存器均被置 1,DMA 请求被禁止。另外,如果一个通道没有设置自动重装功能,那么,一旦 DMA 传送结束, \overline{EOP} 信号有效,会自动置

1 屏蔽触发器。因此对 DMA 通道进行初始化时必须清除屏蔽触发器,方法为:对端口 DMA+0EH 进行一次写操作,即可清除 4 个通道的屏蔽触发器。

例:

```
MOV DX,DMA+0EH      ;DMA 代表 8237A 的片选地址
MOV AL,0
OUT DX,AL
```

9. 先/后触发器

8237A 只有 8 根数据线,而基地址寄存器和基本字节寄存器都是 16 位,预置初值时须分两次进行,每次写入一个字节。

设置先/后触发器是为规定初值的写入顺序。将先/后触发器清零,则初值写入顺序为先写低位字节,后写高位字节。

10. 暂存寄存器

暂存器为 4 通道共用的 8 位寄存器。在 DMA 控制器实现存储器到存储器传送方式时,它暂存中间数据,暂存器中内容 CPU 可以读取,其值为最后一次传送的数据。

8237A 可寻址的内部寄存器,端口地址如表 12-1 所示。

表 12-1 PC/AT DMAC 寄存器 I/O 端口地址(16 进制)

8237A 内部寄存器端口地址	DMAC1	DMAC2	内部寄存器名称
DMA+00H	000	0C0	CH0 基地址寄存器和当前地址寄存器
DMA+01H	001	0C2	CH0 基本字节寄存器和当前字节寄存器
DMA+02H	002	0C4	CH1 基地址寄存器和当前地址寄存器
DMA+03H	003	0C6	CH1 基本字节寄存器和当前字节寄存器
DMA+04H	004	0C8	CH2 基地址寄存器和当前地址寄存器
DMA+05H	005	0CA	CH2 基本字节寄存器和当前字节寄存器
DMA+06H	006	0CC	CH3 基地址寄存器和当前地址寄存器
DMA+07H	007	0CE	CH3 基本字节寄存器和当前字节寄存器
DMA+08H	008	0D0	状态寄存器/控制寄存器
DMA+09H	009	0D2	请求寄存器
DMA+0AH	00A	0D4	屏蔽寄存器
DMA+0BH	00B	0D6	方式寄存器
DMA+0CH	00C	0D8	先/后触发器
DMA+0DH	00D	0DA	暂存器/复位命令
DMA+0EH	00E	0DC	清屏蔽寄存器
DMA+0FH	00F	0DE	多通道屏蔽寄存器

12.2.3 8237A 的时序

图 12-9 是 8237A 的典型工作时序。

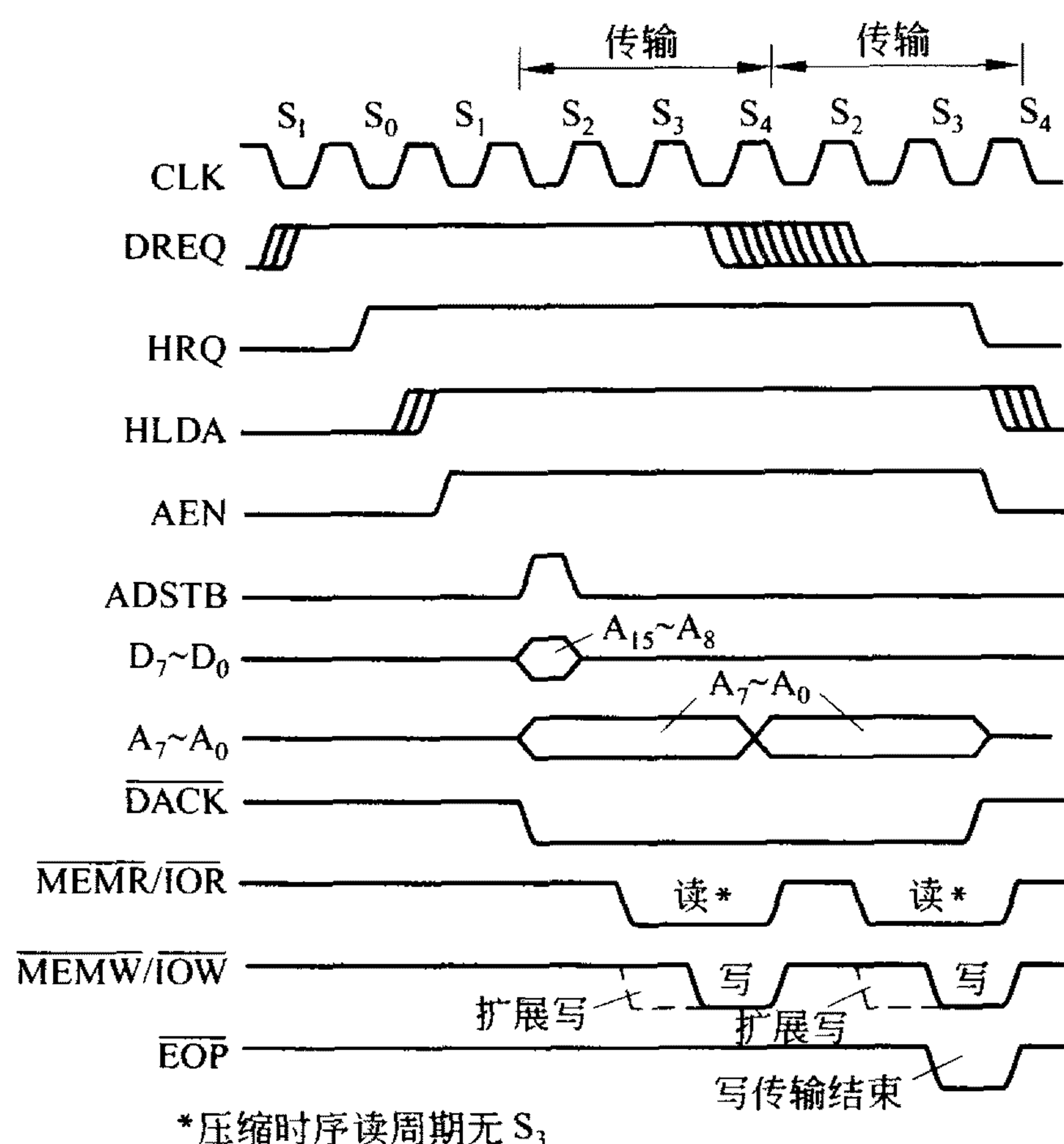


图 12-9 DMA 传输(数据块传送方式)正常时序

8237A 的工作过程可以分为七种状态,即 S_1 、 S_0 、 S_1 、 S_2 、 S_3 、 S_4 和 S_w 。

S_1 : 空闲状态。8237A 初始化完成后即处于 S_1 状态。在 S_1 状态,8237A 不断检测 DREQ,一旦有 DMA 请求信号,8237A 使 HRQ 有效,向 CPU 发出总线请求,随后进入 S_0 状态。

S_0 : 等待状态。不断检测 CPU 发来的总线响应 HLDA 信号,若 HLDA 信号有效,进入 S_1 状态。

S_1 : 使输出端 AEN 有效,利用 AEN 有效,DMA 控制电路把要访问的 RAM 单元 $A_{15} \sim A_8$ 地址送到 $DB_7 \sim DB_0$ 数据线上。

S_2 : 使地址选通信号 ADSTB 有效。ADSTB 的下降沿把 $DB_7 \sim DB_0$ 数据线上的信息锁存到外部的地址锁存器中,并在 S_3 状态出现在地址总线上;同时地址线 $A_7 \sim A_0$ 上输出要访问的 RAM 单元 $A_7 \sim A_0$ 的地址,该信息存放在外围的锁存器中,它被直接送到地址总线上,并且在整个 DMA 传送期间保持住; \overline{DACK} 有效时,通知 I/O 端口做好数据传送的准备。

S_3 : \overline{IOR} 或 \overline{MEMR} 有效,进行读操作。

S_4 : \overline{IOW} 或 \overline{MEMW} 有效,进行写操作。

$S_1 \sim S_4$ 是标准的 DMA 操作周期,在 S_4 状态时检测 \overline{EOP} 信号,若有效,则 DMA 操作结束,8237A 进入 S_1 空闲状态。

说明:

① 在数据块传送方式时, S_4 之后应接着传送下一个字节,一般情况下,地址的高 8 位不变,只是低 8 位进行增址和减址。因此,高 8 位没有必要再送到 $DB_7 \sim DB_0$ 上,也没

有必要再次发出 ADSTB 信号进行锁存,可直接进入 S_2 状态。

② 如果 RAM 或 I/O 端口工作速度比较慢,不能在规定时间内完成数据写入,应设计等待电路,使 Ready 信号为低,8237A 在 S_3 后沿检测到 Ready 信号为低时,自动插入一个等待状态 S_w ,在 S_w 状态, S_3 状态的所有控制信号都不变,只是延长读、写时间,直到 Ready 信号变为高电平,进入 S_4 状态。

③ 关于“扩展的写信号”。8237A 初始化编程时,如果控制寄存器 D_5 位置 1,那么在每个 DMA 周期的 S_3 状态,将提前出现有效的“写”信号,即“扩展的写信号”。

12.3 8237A 的应用

12.3.1 8237A 的初始化编程

1. 命令字写入控制寄存器

初始化时必须设置控制寄存器,以确定其工作时序、优先级方式、DREQ 和 DACK 的有效电平及是否允许工作等。

在 PC 系列机中,BIOS 初始化时,已将通道的控制寄存器设定为 00H,禁止存储器到存储器传送,允许读/写传送,正常时序,固定优先级,不扩展写信号,DREQ 高电平有效,DACK 低电平有效,因此在 PC 微型计算机系统中,如果借用 DMA CH1(CH1 是预留给用户使用的)进行 DMA 传送,则初始化编程时,不应再向控制寄存器写入新的命令字。

2. 屏蔽字写入屏蔽寄存器

某通道正在进行初始化编程时,接收到 DMA 请求,可能未初始化结束,8237A 就开始进行 DMA 传送,导致出错。因此初始化编程时,必须先屏蔽要初始化的通道,初始化结束后,再解除该通道的屏蔽。

3. 方式字写入方式寄存器

为通道规定传送类型及工作方式。

4. 置 0 先/后触发器

对端口地址 $DMA+0CH$ 执行一条输出指令(写入任何数据均可),从而产生一个写命令,即可置 0 先/后触发器,为初始化基地址寄存器和基本字节寄存器做准备。

5. 写入基地址和基本字节寄存器

把 DMA 操作所涉及到的存储区首地址或末地址写入基本地址寄存器,把要传送的字节数减 1,写入基本字节寄存器。这几个寄存器都是 16 位的,因此写入要分两次进行,先写低 8 位(则先/后触发器置 1),后写高 8 位(则先/后触发器自动置 0)。

6. 解除屏蔽

初始化之后向通道的屏蔽寄存器写入 $D_2 \sim D_0 = 0 \times \times$ 的命令字,置 0 相应通道的屏蔽触发器,准备响应 DMA 请求。

7. 写入请求寄存器

如果采用软件 DMA 请求,在完成通道初始化之后,在程序的适当位置向请求寄存器写入 $D_2 \sim D_0 = 1 \times \times$ 的命令字,即可使相应通道进行 DMA 传送。

12.3.2 8237A 在 IBM PC/AT 系统中的应用

IBM PC/AT 使用两片 8237A 级联,提供 7 个 DMA 通道,通道 0~通道 3 支持 8 位数据传送,通道 5~通道 7 支持 16 位数据传送,PC/AT 有专门的动态 RAM 刷新电路,硬盘驱动器采用高速 PIO 传送,无须 DMA 支持,通道 1 给用户使用,通道 2 服务于软盘驱动器,通道 4 作为两个 DMA 控制器的级联,其余均保留备用。

PC/AT DMAC 寄存器 I/O 端口地址见表 12-1 所示。

PC/AT 地址总线的宽度是 24 位,由 $A_{23} \sim A_0$ 组成,最大寻址空间可达 16MB。

由于 8237A 内部地址寄存器是 16 位,只能寻址 64KB 空间,如何扩大 8237A 的寻址空间? PC/AT 微型计算机系统在 8237A 芯片以外为每一通道设置一个 8 位的页面寄存器,如图 12-10 所示。

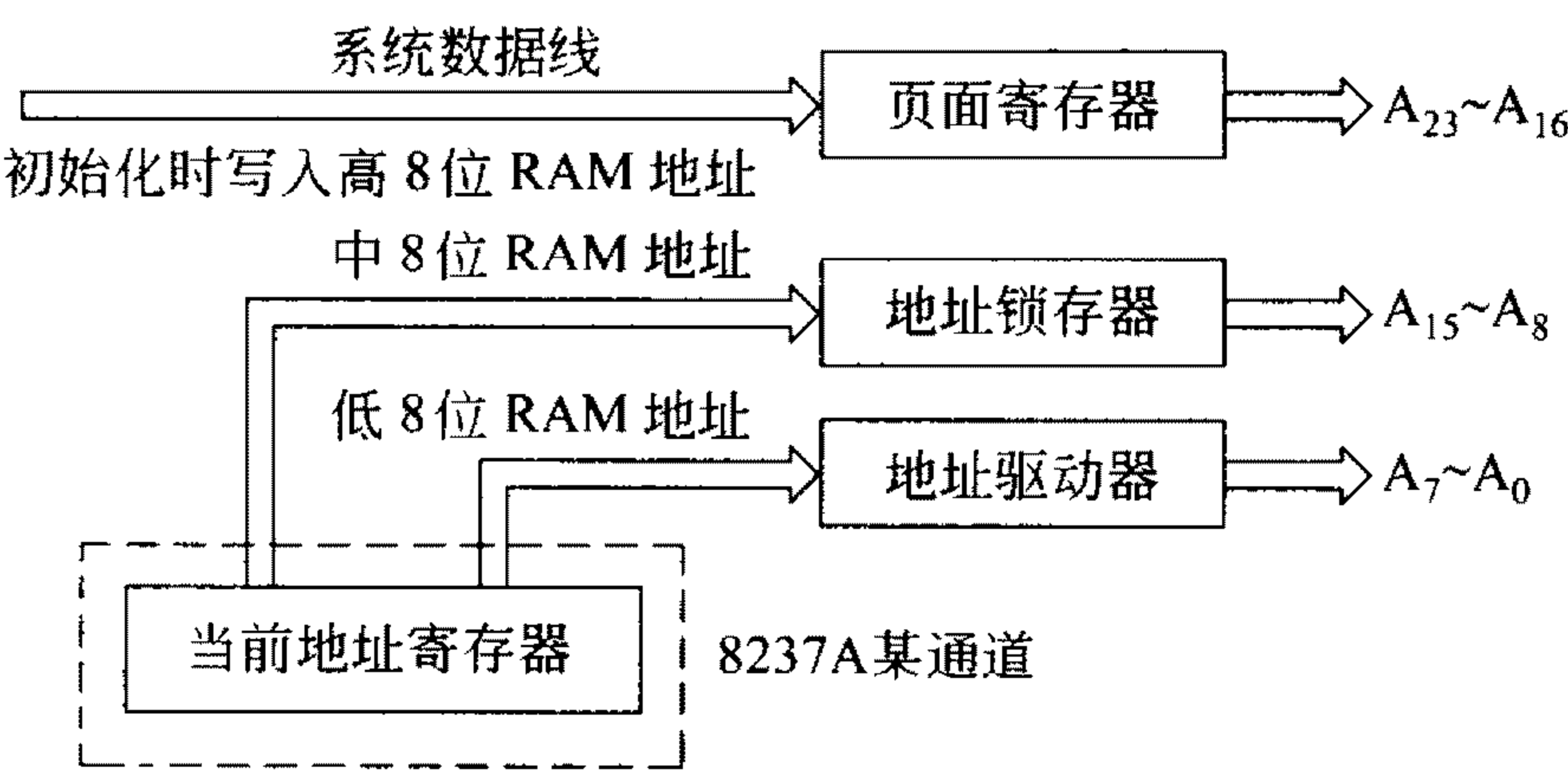


图 12-10 扩大 8237A 寻址范围示意图

在进行 DMA 读/写传送之前,程序要把 DMA 传送所涉及到的 RAM 单元的高 8 位物理地址写入到相关通道的页面寄存器,把 RAM 单元低 16 位物理地址写入到相关通道的基本地址寄存器,把 DMA 传送的实际字节数减 1,写入相关通道的基本字节寄存器,从而做好初始化准备。

一旦 I/O 端口有 DMA 请求,并且 DMAC 控制系统的三总线之后,由相关通道的 DMA 应答信号控制把页面寄存器内容送到地址总线高 8 位,DMAC 把相关通道的低 16 位地址经过外部地址锁存和驱动送到低 16 位地址总线上,选择某一存储单元。

在 PC/AT 系统中,页面寄存器采用专用的三态输出存储器映像器 74LS612 来实现,在高档微型计算机中,DMAC 和相关页面寄存器都被兼容的多功能芯片所取代。页面寄

存器端口地址如表 12-2 所示。

表 12-2 PC/AT 页面寄存器的 I/O 端口地址

DMAC 通道	I/O 地址	DMAC 通道	I/O 地址
CH0	87H	CH6	89H
CH1	83H	CH7	8AH
CH2	81H	存储器刷新	8FH
CH3	82H	错误标志单元	80H
CH5	8BH		

习 题

1. DMA 系统完成的功能是什么？
2. DMA 传送方式和中断方式相比，各有什么特点？
3. 8237A 的主要功能是什么？
4. 8237A 内部寄存器各有什么作用？
5. 80286 系统一个存储单元是 24 位物理地址，而 8237A 在寻址内存空间时，只能给出 16 位地址码，这一矛盾如何解决？有哪些硬件和软件措施？
6. 8237A 提供哪几种传送方式？在微型计算机系统中，不允许使用哪一种传送方式？
7. 8237A 初始化编程的步骤是什么？
8. 简述 PC 系列机用 DMA 方式进行单字节读写传送的全过程。
9. 8237A 芯片采用数据块传送方式和单字节传送方式进行 DMA 传输时，其主要区别在哪里？
10. 什么是 DMA 控制器的正常时序和压缩时序？

数模和模数转换

计算机用于数据采集的时候,要采集的量往往是连续变化的物理量,但计算机只能处理数字量,因此需要用专门的器件把模拟量转换成数字量再送交计算机处理。在过程控制的时候,如果控制对象需要模拟量,就需要用专门的器件把计算机输出的数字量转换成模拟量,再去控制受控器件。A/D 转换器完成模拟量到数字量的转换,而 D/A 转换器完成数字量到模拟量的转换。

13.1 数模转换

13.1.1 数模转换原理

数/模转换器件的核心是“解码网络”,常用的解码网络有“权电阻解码网络”,“T 型解码网络”等等,本节仅就 T 型解码网络介绍数/模转换原理。

图 13-1 是一个 8 位数字量→模拟量的 T 型解码网络示意图,图中 VREF 为参考电压,内阻为 0,求和元件是外接的运算放大器,内阻视为 0, S_7 、 S_6 … S_0 为 8 个电子开关,它们受控于待转换数字量的各位二进制代码,设待转换数字量的数列为:

$$N = D_7 D_6 \cdots D_0 \quad (D_7 \text{ 为最高位})$$

则 S_7 受控于 D_7 位, S_6 受控于 D_6 位……。若 D_7 位等于“1”,则 S_7 倒向右侧与参考电压接通;若 D_7 位等于“0”,则 S_7 倒向左侧与地接通,以此类推。

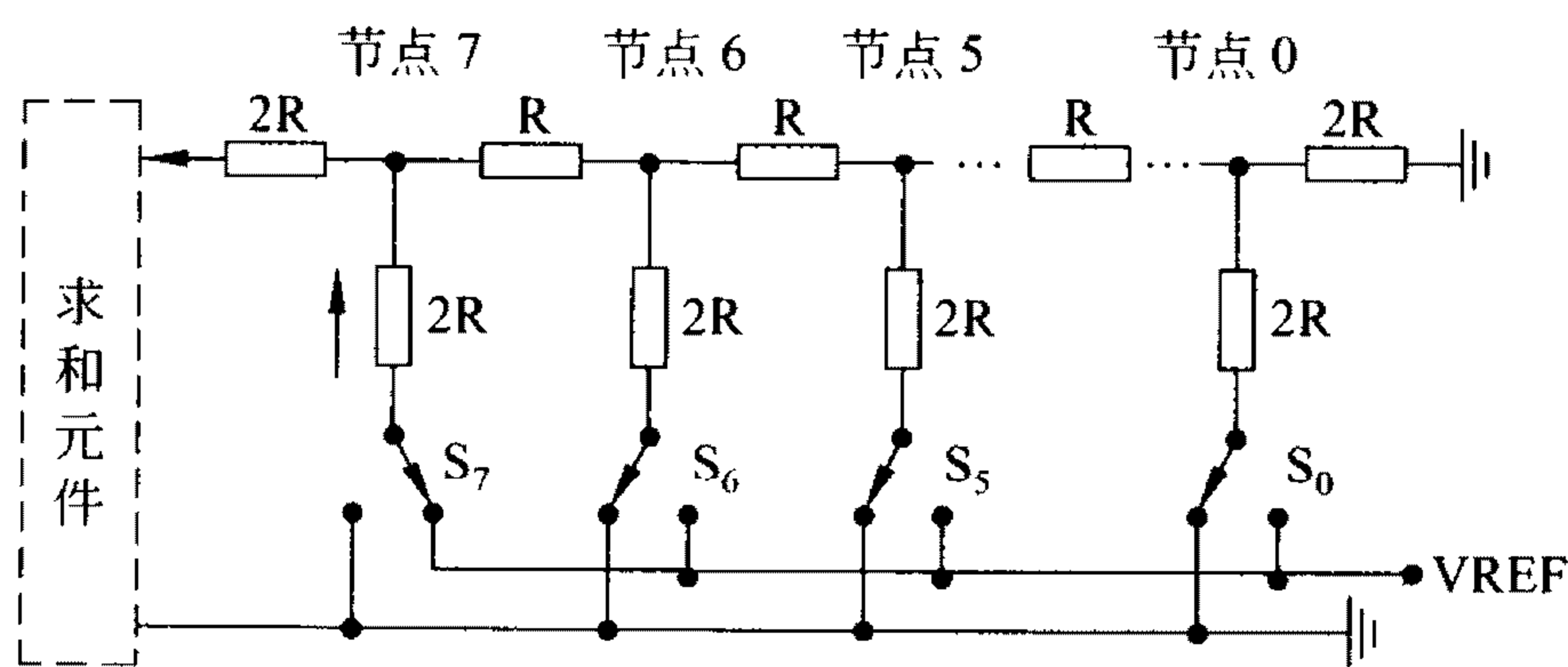


图 13-1 T 型解码网络 D/A 转换器

解码网络相邻两节点之间的电阻都为 R , 但节点 7 与求和元件之间, 节点 0 与地之间的电阻为 $2R$, 各支路电阻为 $2R$ 。不论电子开关倒向何方, 从任一节点向左、向右看去(不包含支路电阻本身)的等效电阻都是 $2R$ 。

我们用叠加原理来分析输入到求和元件的总电流。方法是: 依次假设 $S_7 \sim S_0$ 中只有一个开关接通 V_{REF} , 其他开关都接地, 这相当于待转换的数字量(二进制数)中, 只有一位为“1”, 其他位都为“0”。在这种情况下, 求出接通 V_{REF} 的支路所产生的总电流的分量是多少, 然后把各种情况下的总电流的分量叠加起来, 就是 8 位数字量在 256 种组合状态下产生的总电流了。

在如图 13-1 所示的 T 型网络中, 只有一个开关接 V_{REF} , 其他开关都接地, 此时接通 V_{REF} 的支路, 其支路电流总是 $V_{REF}/3R$ (假设 $V_{REF}/3R = I$), 但是各个支路电流向求和元件提供的总电流分量是不同的, 权值最高的支路(S_7 支路), 其支路电流提供的总电流分量最大。

当 S_7 接 V_{REF} , 其他开关接地, S_7 支路电流提供的总电流的分量为 $I/2$ 。

当 S_6 接 V_{REF} , 其他开关接地, S_6 支路电流提供的总电流的分量为 $I/4$ 。

.....

当 S_0 接 V_{REF} , 其他开关接地, S_0 支路电流提供的总电流的分量为 $I/256$ 。

当 8 位数字量 $= 0 \sim 255$ 变化时, 各支路电流产生的总电流的分量之和就是流向求和元件的总电流。

$$\begin{aligned} \text{总电流} &= D_7 \times I \times 2^{-1} + D_6 \times I \times 2^{-2} + \dots + D_0 \times I \times 2^{-8} \\ &= I \times 2^{-8} (D_7 \times 2^7 + D_6 \times 2^6 + \dots + D_0 \times 2^0) \\ &= N \times 2^{-8} \times (V_{REF}/3R) \end{aligned}$$

由于 V_{REF} 、 $3R$ 均为固定的值, 所以总电流只和输入的二进数 N 有关, 通过解码网络之后得到了一个与输入的数字量成比例关系的电流。

集成化的 D/A 转换芯片按分辨率可分为 6、8、10、12、14、16 位 D/A 转换器, 大多数芯片内部配有输入数据寄存器, 可以和 CPU 的数据总线相连, 而没有输入数据寄存器的芯片, 不能直接和 CPU 数据线相连, 求和元件通常是外接的运算放大器。

13.1.2 DAC 0832 简介

1. 引脚和内部结构

DAC 0832 是 8 位的 D/A 转换芯片, 引脚与内部结构如图 13-2 所示。

DAC 0832 内部有两级输入缓冲寄存器和一个 T 型解码网络, 是电流输出型芯片, 解码网络输出需要外接运算放大器才能得到模拟电压输出, 由于输入有缓冲寄存器, 所以 DAC0832 的数据线可以和微处理器的数据线直接相连。

$DI_7 \sim DI_0$: 8 位数字量输入数据线, DI_7 为最高位。

\overline{CS} : 片选信号输入端。

ILE : 输入锁存允许信号。

$\overline{WR1}$: 写信号 1。

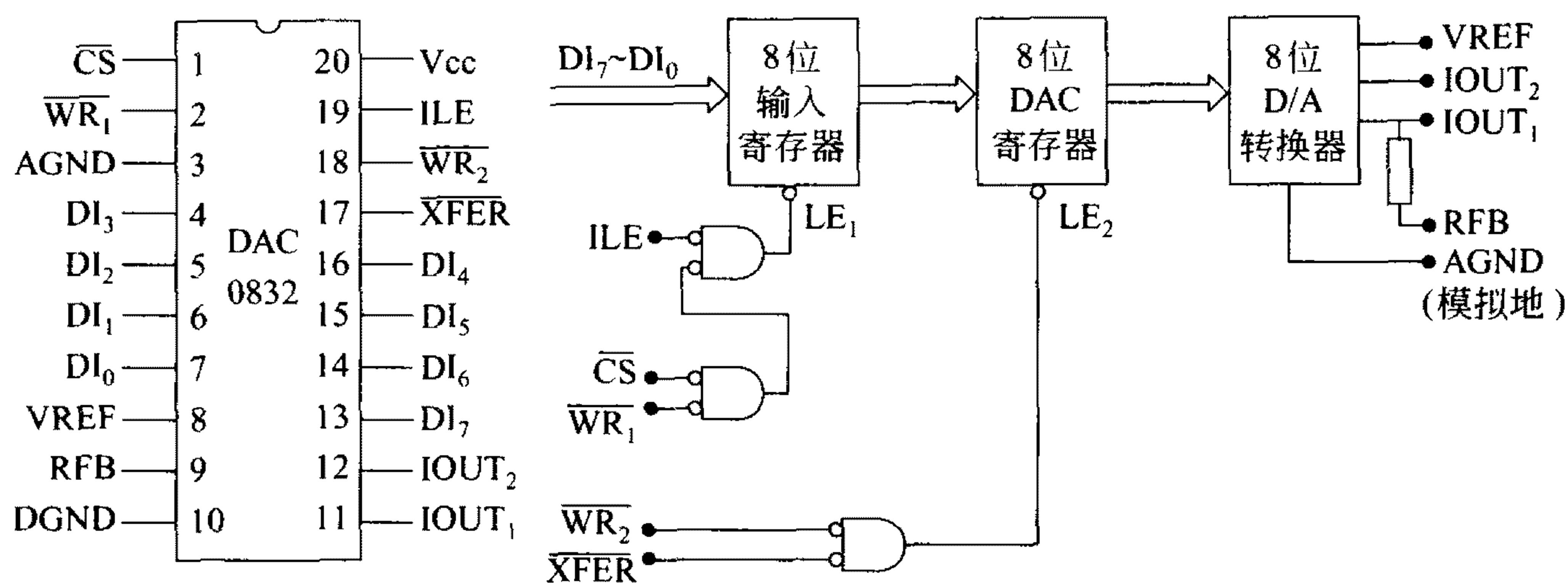


图 13-2 DAC 0832 引脚及内部结构

\overline{WR}_2 : 写信号 2。

\overline{XFER} : 传送控制信号。

$IOUT_1$: 模拟电流输出端 1, 当输入数据为全 1 时, 输出电流最大, 当输入数据为全 0 时, 输出电流为 0。

$IOUT_2$: 模拟电流输出端 2 ($IOUT_1 + IOUT_2 = \text{常数}$)。

RFB: 内部反馈电阻引出端, 外部的运算放大器输出端可以直接接到 RFB 端。

VREF: 外接参考电压, 电压范围: $-10V \sim +10V$ 。

Vcc: 芯片工作电压, 电压范围: $+5V \sim +15V$ 。

AGND: 模拟信号地。

DGND: 数字信号地。

2. 工作方式

DAC 0832 内部有两级输入缓冲寄存器, 当 LE_1 为高电平时 ($ILE=1, \overline{CS}=0, \overline{WR}_1=0$), 输入寄存器的输出端信号跟随 $DI_7 \sim DI_0$ 变化, 当 LE_1 由高 \rightarrow 低时 ($ILE=0$, 或 $\overline{CS}=1$, 或 $\overline{WR}_1=1$), 输入寄存器锁存 $DI_7 \sim DI_0$ 的当前值。当 LE_2 为高电平时 ($\overline{WR}_2=0, \overline{XFER}=0$), DAC 寄存器的输出信号跟随输入寄存器的输出信号变化, 当 LE_2 由高 \rightarrow 低时 ($\overline{WR}_2=1$ 或 $\overline{XFER}=1$), DAC 寄存器锁存当前的输入寄存器输出值, 送 D/A 转换器进行转换, 因此 DAC 0832 有 3 种工作方式:

(1) 双缓冲方式

通常采用的接线方式为: ILE 固定接 $+5V$, CPU 的 \overline{IOW} 信号复连到 \overline{WR}_1 和 \overline{WR}_2 , 用 \overline{CS} 作为输入寄存器的“片选”信号, \overline{XFER} 作为 DAC 寄存器的“片选”信号, 分别接到两个 I/O 端口地址译码输出。

数据写入分两次进行, 第 1 次对输入寄存器写入待转换的数字量, 第 2 次对 DAC 寄存器执行一次写操作, 第 2 次的写操作只是一次“虚拟写操作”, 写入什么数据是无关紧要的, 因为它不能存入输入寄存器中, 目的只是为了启动 DAC 寄存器的锁存功能。

双缓冲方式的优点是: 在 D/A 转换的同时, 可以接收下一个待转换的数据, 从而提

高转换速率。

(2) 单缓冲方式

采用单缓冲方式是令两个寄存器中的一个处于直通状态,例如把 \overline{WR}_2 , \overline{XFER} 接地(数字信号地),使DAC寄存器处于直通状态, \overline{ILE} 接+5V, \overline{WR}_1 接CPU的 \overline{IOW} , \overline{CS} 接I/O端口地址译码器。只针对 \overline{CS} 端进行一次写入操作,数据写入后即开始D/A转换。

(3) 直通方式

当 \overline{ILE} 接+5V, \overline{CS} , \overline{WR}_1 , \overline{WR}_2 , \overline{XFER} 都接地(数字信号地)时,DAC 0832处于直通方式,输入端 $DI_7 \sim DI_0$ 一旦出现数字信号就立即进行D/A转换,由于输入不使用缓冲寄存器,所以不能和计算机系统的数据线相连。

在进行D/A(A/D)转换时,要正确处理芯片的接地问题。D/A转换器(A/D转换器)的内部主要是模拟电路,而运算放大器内部则完全是模拟电路,在设计D/A(A/D)转换电路时,必然要有其他的门电路、译码器等协同工作,它们是数字电路,因此就存在“模拟”和“数字”两个系统,为了避免干扰,应当把系统的“数字地”连在一起,“模拟地”连在一起,然后再把数字地和模拟地连接到一个共地点。在使用DAC 0832时, \overline{WR} 信号的宽度应不小于500ns,待转换的数字量保持时间应在90ns以上。

13.2 模数转换

13.2.1 模数转换原理

模/数(A/D)转换的方法比较多,集成化的A/D转换芯片通常采用逐次逼近法,为了介绍逐次逼近法的原理,我们先介绍计数式A/D转换的原理。

1. 计数式A/D转换原理

图13-3为8位计数式A/D转换原理图,当给出一个启动信号之后,计数器清零,产生的8位数字量经内部的D/A转换输出零伏电压,此时比较器输出高电平,计数器在

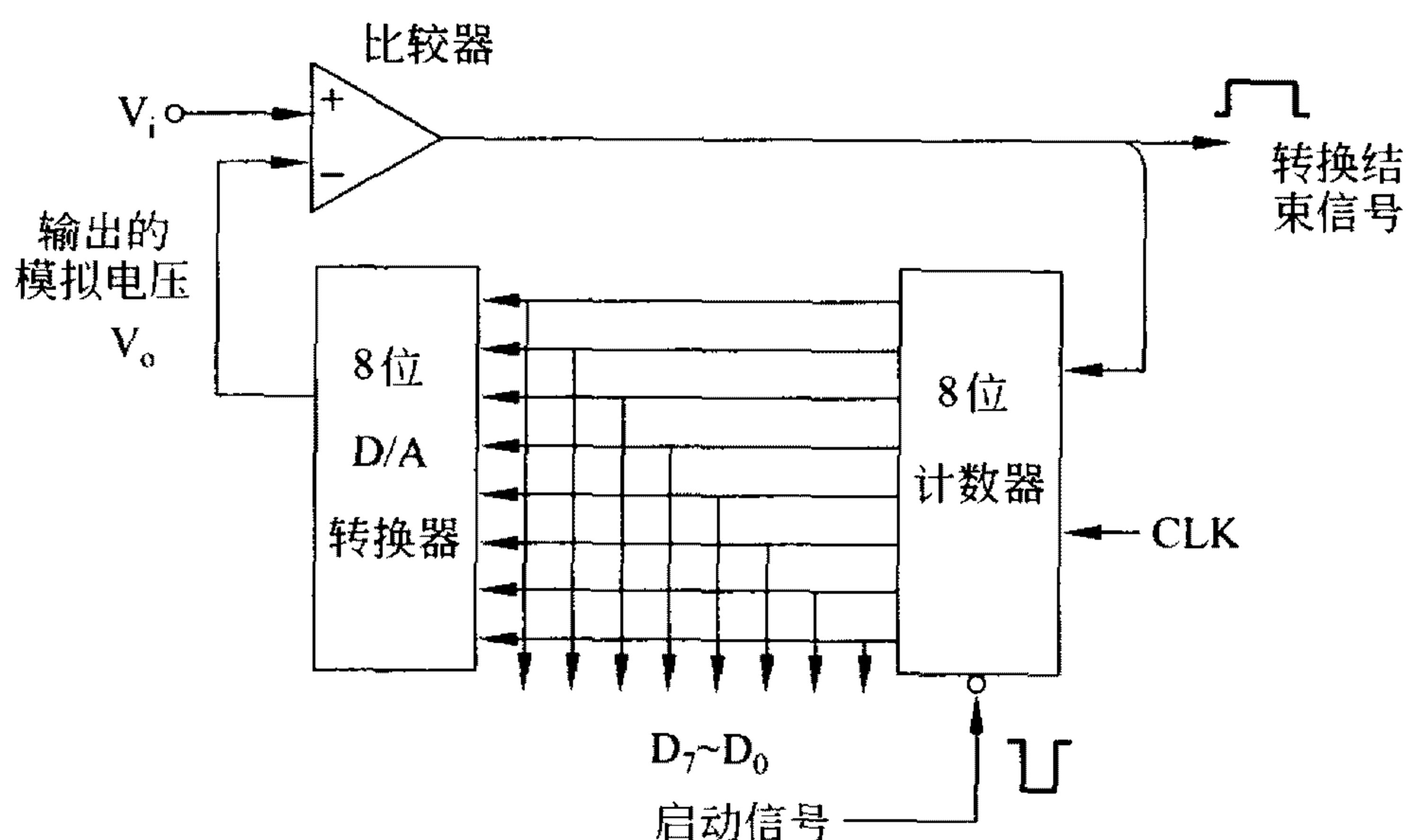


图 13-3 8位计数式 A/D 转换原理

CLK信号驱动下加1计数,使D/A转换后的电压不断上升,当D/A转换后的电压等于输入的模拟电压时,比较器输出为0,使计数器停止计数,数据线 $D_7 \sim D_0$ 上的数字量就是模/数转换的结果。

计数式A/D转换的缺点是速度慢,特别是输入电压较高时,转换速度更慢,如果最高输入电压为5伏,用8位计数器需要255个计数周期才能计到255,用12位计数器(即12位A/D转换器)需要4095个计数周期才能计到4095完成A/D转换。

2. 逐次逼近式 A/D 转换

图13-4为逐次逼近式A/D转换原理图,它的最大特点是用“逐次逼近寄存器”取代了计数式A/D转换中的“加1计数器”。

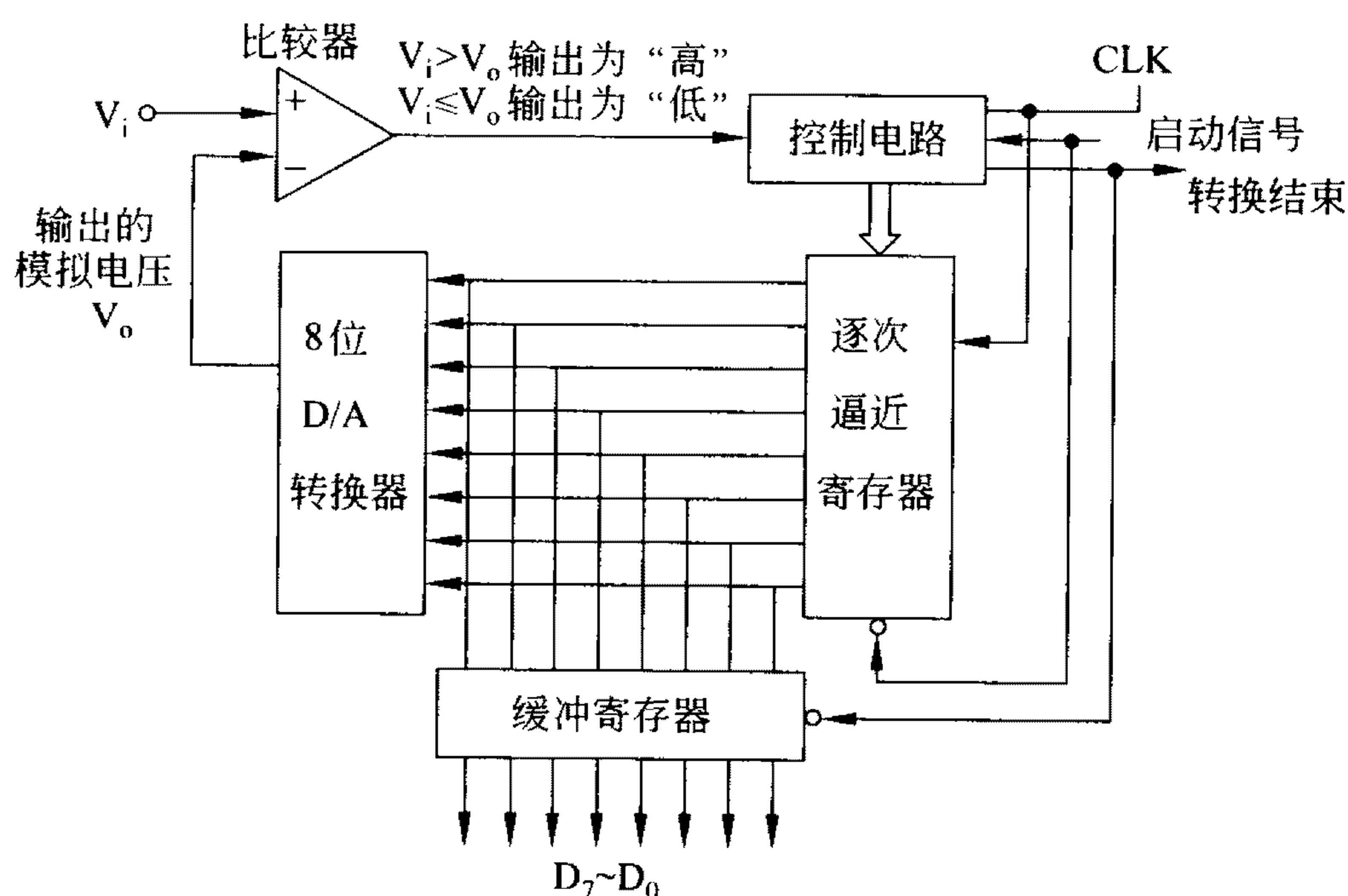


图 13-4 逐次逼近式 A/D 转换原理

当转换器收到启动信号之后,逐次逼近寄存器清零,通过内部的D/A转换器使输出电压 V_o 为0,当启动信号结束后开始转换。

第1个CLK周期,控制电路使逐次逼近寄存器最高位为1(即10000000)这一组数字经过内部D/A转换之后,产生一个 V_o ,如果 $V_i > V_o$,比较器输出为“高”,通过控制电路使刚才的置1位保留下来。第2个CLK周期,再使次高位为1(即11000000),如果11000000产生的 V_o 比 V_i 大,则比较器输出为“低”,通过控制电路使刚才的置1位(D_6 位)清零,接下来再使 D_5 位置1……,重复上述过程,直到 D_0 位试探完毕。

逐次逼近寄存器从最高位开始逐位设置试探值,转换完毕,逐次逼近寄存器中的值经缓冲器输出,其转换速度比计数式A/D转换快的多,集成电路的A/D转换芯片大多采用这种方式。

13.2.2 ADC 0809 简介

ADC 0809 引脚及内部结构如图13-5所示。

ADC 0809 是逐次逼近式的模/数转换芯片,ADC 0809 芯片可以完成8路模拟量→

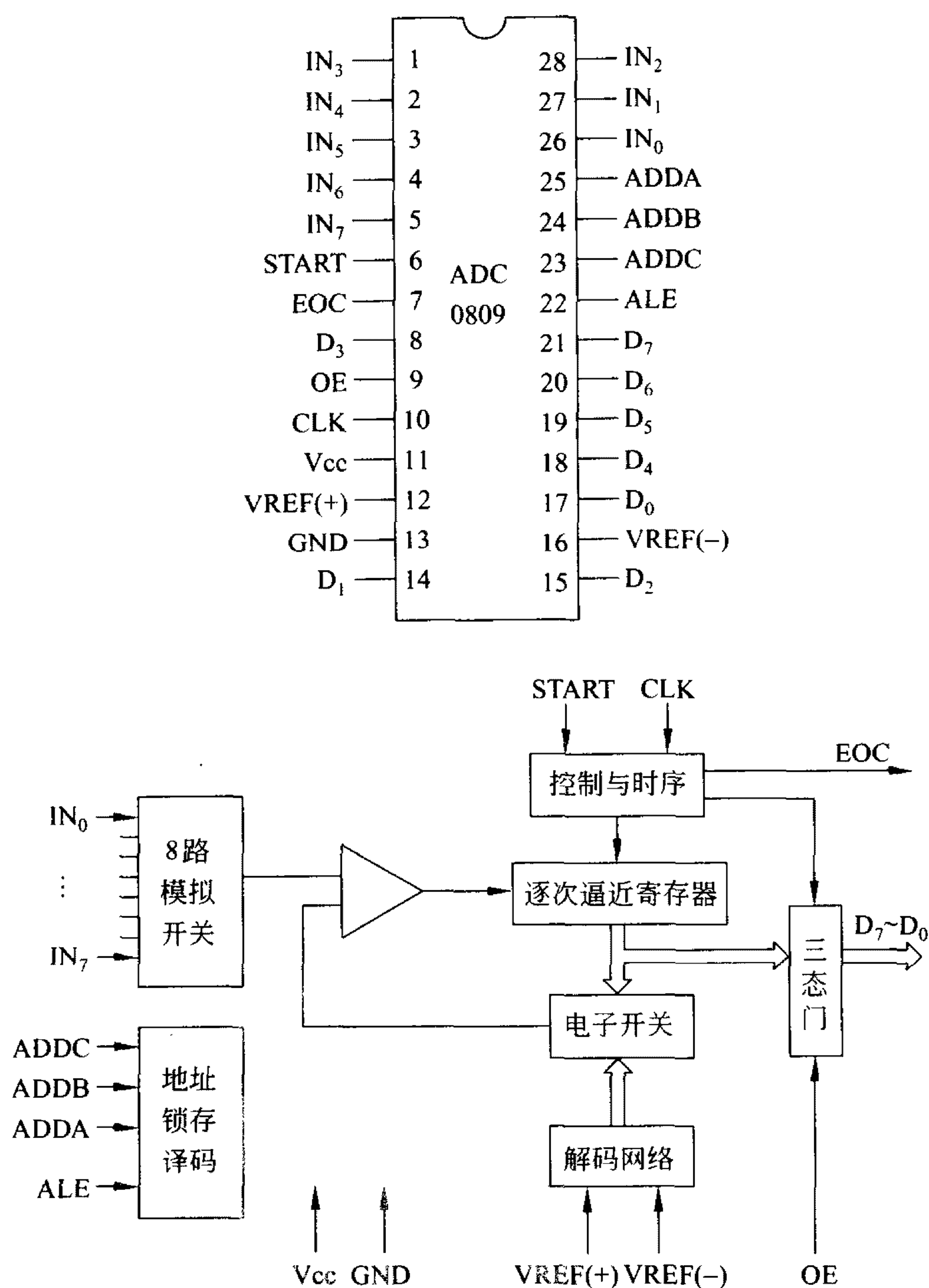


图 13-5 ADC 0809 引脚图及内部结构

数字量的转换,内部配有地址译码电路,通过地址线 ADDC、ADDB、ADDA 和地址锁存信号 ALE,选通 IN₀~IN₇ 这 8 路模拟量之一。内部采用逐次逼近的 A/D 转换原理,转换后的 8 位数字量通过三态缓冲器输出,因此可以和微处理器的数据线直接相连。一次模拟量的转换时间为 100 μ s,转换结束后从 EOC 端输出转换结束信号,如果 ADC 0809 用于微处理器系统,EOC 信号可作为 CPU 的中断请求信号。

CLK:	时钟输入信号,最大 640kHz。
IN ₀ ~IN ₇ :	8 路模拟量输入。
ADDC、ADDB、ADDA:	模拟通道选择信号。
ADDC,ADDB,ADDA	=000 选择 IN ₀ 。
	=001 选择 IN ₁ 。

	=111 选择 IN ₇ 。

ALE: 地址锁存允许信号, ALE 有效时锁存 ADDC~ADDA 的通道选择信号。

START: A/D 转换启动信号, 高电平时内部的逐次逼近寄存器清零, 由高→低时开始转换。START 常与 ALE 相连, 以便同时锁存通道选择信号, 并开始 A/D 转换。START、ALE 信号宽度不小于 100ns。

EOC: 转换结束信号, EOC 由低→高表示转换结束, 当 ADC 0809 用于微型计算机系统时, EOC 可作为 CPU 的中断请求信号。

OE: 输出允许信号, OE 有效时, 打开输出三态门, 输出转换后的数字量。

$D_7 \sim D_0$: 输出数据线。

VREF(+), VREF(-): 参考电压+5V。

V_{CC} : 工作电压+5V。

习 题

1. 简述 DAC 0832 双缓冲方式的工作原理。
2. 从 DAC 0832 的 $IOUT_1$ 和 $IOUT_2$ 输出的是与输入数字量成正比关系的模拟电流, 若要得到模拟电压怎么办?
3. 简述逐次逼近式的 A/D 转换原理。
4. 如果 ISA 总线外扩一片 ADC 0809, 采集 8 路模拟量, 通道选择信号 ADDA~ADDC 应怎样连接? 为了启动转换和读取转换后的数字量, 还需要设计怎样的外围电路?

保护模式及其编程

32 位处理器有两种工作模式：实地址模式(简称实模式)和保护虚拟地址模式(简称保护模式)。为了在保护模式下能够继续运行 8086 的应用程序,32 位处理器又提供了虚拟 86 模式。

32 位处理器只有在保护模式下,才能真正发挥其最大作用。在保护模式下,存储器的分段和分页管理机制,不仅为存储器共享和保护提供了硬件支持,而且为实现虚拟存储器提供了硬件支持;保护模式支持多任务,能够快速地进行任务切换,并保护任务环境;它还提供了 4 个特权级,并配合以完善的特权检查机制,既能实现资源共享,又能保证代码、数据的安全和保密及任务的隔离。保护模式是 32 位处理器的主要工作模式,Windows、Linux 和 UNIX 等多任务操作系统都工作在 80x86 处理器的保护模式。本章介绍保护模式的主要内容及其相关的程序设计。

14.1 保护模式下的存储管理

本书的第 2 章已经介绍过,32 位处理器有三个明确的存储地址空间,分别是虚拟空间(又称逻辑空间,编程空间)、线性空间和物理空间(又称主存空间、实际空间)。相应的地址称为虚拟地址(又称逻辑地址)、线性地址和物理地址(又称主存地址)。

保护模式下,程序员编程采用段选择子和段内偏移构成的二维虚拟地址来访问存储器,32 位处理器支持的虚拟地址空间可达 64TB。因此可以认为有足够大的存储空间供编程使用。

显然,只有在物理存储器中的代码和数据才能被 CPU 执行和访问,因此,虚拟地址空间必须映射到物理地址空间,二维的虚拟地址必须转化成一维的物理地址。大多数的 32 位处理器有 32 条地址线(Pentium Pro 有 36 条地址线),在保护模式下,可寻址访问的物理地址空间的范围为 4GB(Pentium Pro 为 64GB)。

在多任务系统中,每一个任务有一个虚拟地址空间。为了避免多个并行任务的多个虚拟地址空间直接映射到同一个物理地址空间,采用线性地址空间隔离虚拟地址空间和物理地址空间。线性地址空间由一维的线性地址构成,线性地址是 32 位,线性地址空间范围为 4GB。

在保护方式下,32 位处理器首先采用存储器分段管理机制,将虚拟地址转换为线性

地址;然后提供可选的存储器分页管理机制,将线性地址转换为物理地址。

14.1.1 分段管理

程序员在编写程序时,会定义数据段、代码段等多个逻辑段,分段管理机制的用途就是管理这些段,并将二维的虚拟地址转化成一维的线性地址。

1. 存储段描述符

分段管理后,系统必须知道每个段的段信息才能完善地管理各段。每个段的段信息包括段基址(Base Address)、段界限(Limit)和段属性(Attributes)三个部分。32 位处理器把每个段的段信息放入一个数据结构中,称为描述符。每个描述符长 8 个字节。按描述符所描述的对象来划分,描述符可分为三类:存储段描述符、系统段描述符和门描述符,它们由描述符格式属性字节中的描述符类型位 DT 的值来区分。下面首先介绍存储段描述符,存储段描述符存放可由程序直接进行访问的代码段及数据段。格式如图 14-1 所示。

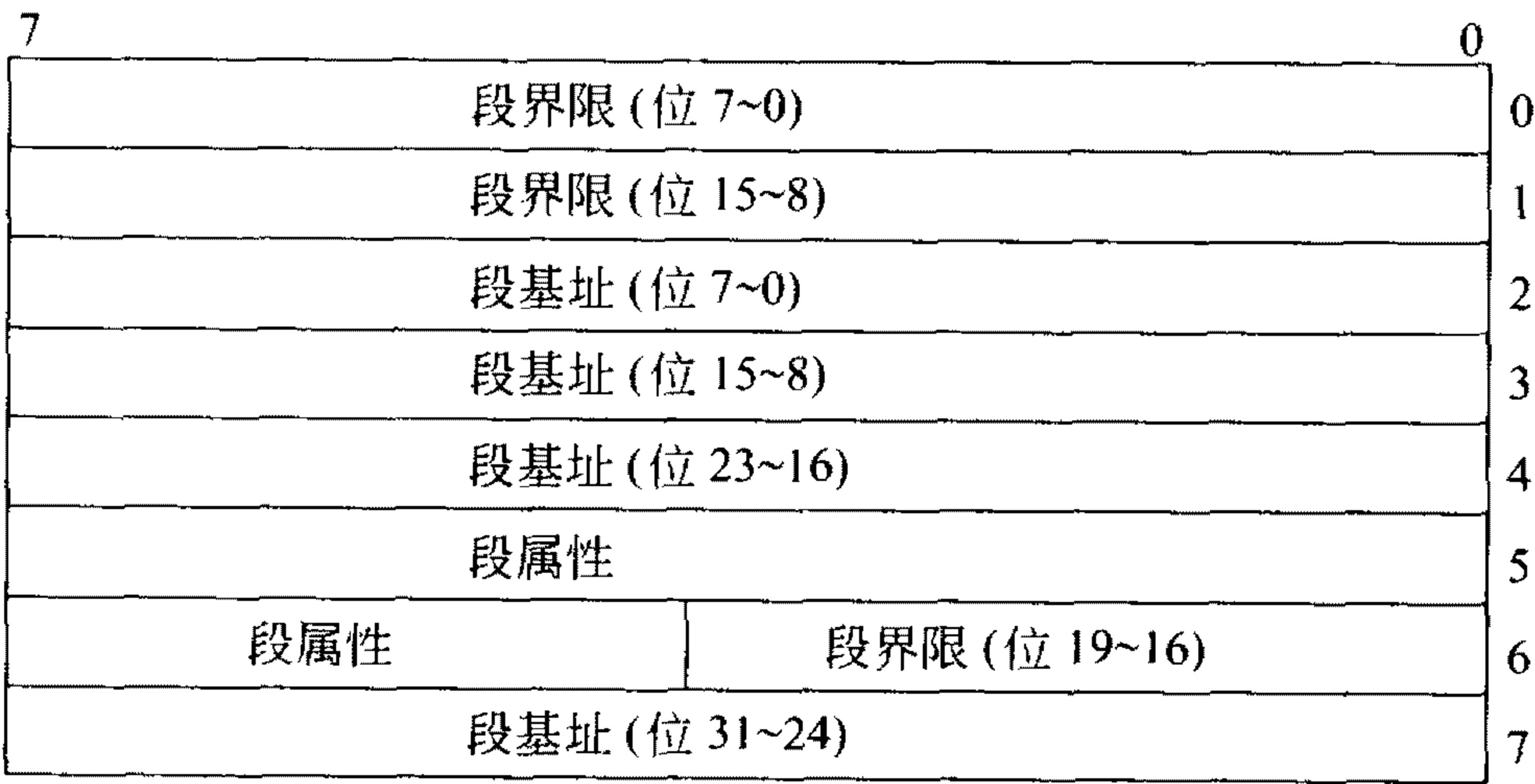


图 14-1 存储段描述符的格式

段基址规定段在线性地址空间中的开始地址。在保护模式下,段基址长度为 32 位。任何一个段,都可以从 32 位线性地址空间中的任何一个字节开始。而实地址模式下规定的段边界必须能被 16 整除。

段界限规定段的大小。在保护模式下,段界限用 20 位表示,而且段界限可以是以字节为单位或以 4KB 为单位。段属性中的 G 位对此进行定义。若段界限以字节为单位,一个段的最大长度为 2^{20} 字节,即 1MB;而以 4KB 为单位,一个段的最大长度为 $2^{20} \times 2^{12}$ 字节,即 4GB。

段属性规定段的主要特性。被存放在第 5 字节和第 6 字节的高四位。具体规定如图 14-2 所示。

- ① P: 存在(Present)位。P=1 表示该描述符所描述的段在内存中;P=0 表示该描述符所描述的段未装入内存中,使用该描述符进行内存访问时会引起异常。
- ② DPL: 描述符特权级(Descriptor Privilege Level),共 2 位。它规定了所描述段的特权级。DPL 用于特权检查,以决定对该段能否访问。

位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0	第 5 字节
P	DPL		DT=1	TYPE			A	
位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0	第 6 字节
G	D	0	AVL	段界限 (位 19~16)				

图 14-2 存储段描述符的属性

③ DT: 描述符类型位。DT=1 表示该段是存储段, 该描述符为存储段描述符; DT=0 是系统段描述符和门描述符。

④ TYPE: 存储段的类型和访问权限。各位的具体定义如表 14-1 所示。

表 14-1 存储段描述符属性中的 TYPE 定义

数据段 (包括堆栈段)			说 明	代 码 段			说 明
E	ED	W		E	C	R	
0	0	0	只读, 向高地址扩展	1	0	0	只执行, 普通代码段
0	0	1	可读/写, 向高地址扩展	1	0	1	可执行/读, 普通代码段
0	1	0	只读, 向低地址扩展	1	1	0	只执行, 一致代码段
0	1	1	可读/写, 向低地址扩展	1	1	1	可执行/读, 一致代码段

⑤ A: 访问位 (Accessed)。A=0 表示该段尚未被访问, A=1 表示该段已被访问。当把描述符的相应段选择子装入到段寄存器时, 处理器把该位置为 1, 表明该段已被访问。

⑥ G: 段界限粒度 (Granularity) 位。G=0 表示段界限以字节为单位; G=1 表示段界限以 4KB 为单位。

⑦ D: 在描述可执行段的描述符中, D=1 表示默认情况下指令使用 32 位地址及 32 位或 8 位操作数, 这样的代码段也称为 32 位代码段; D=0 表示默认情况下, 使用 16 位地址及 16 位或 8 位操作数, 这样的代码段也称为 16 位代码段。

⑧ AVL: 软件可利用位。保留给操作系统或应用程序使用。

2. 全局描述符表和局部描述符表

一个任务允许包含有多个段, 每个段用一个段描述符描述。为了便于组织管理, 这些段描述符顺序存放在线性空间中的指定位置, 组成一个段描述符表。在 32 位处理器中, 有三种类型的描述符表: 全局描述符表 GDT (Global Descriptor Table)、局部描述符表 LDT (Local Descriptor Table) 和中断描述符表 IDT (Interrupt Descriptor Table)。在整个系统中, 全局描述符表 GDT 和中断描述符表 IDT 只有一个; 而局部描述符表, 每个任务可以有一个。这里先介绍 GDT 和 LDT。

(1) 全局描述符表 GDT

全局描述符表 GDT 含有每一个任务都可能或可以访问的段的描述符, 通常包含操

作系统所使用的代码段、数据段和堆栈段的描述符,也包含多种特殊数据段的描述符,如各个 LDT 的描述符等。每个 GDT 最多含有 8192 个描述符。注意: GDT 的第 0 个描述符总不被处理器访问,通常它置成全 0。

(2) 局部描述符表 LDT

保护模式支持多任务,每个任务都有自己的局部描述符表 LDT,且每个任务最多只有一个 LDT。每个任务的 LDT 含有该任务自己的代码段、数据段和堆栈段的描述符。每个 LDT 最多含有 8192 个描述符。

3. 段选择子

在实地址模式下,逻辑地址由 16 位段基址和 16 位段内偏移地址两部分组成。段基址存放在段寄存器中。而在保护方式下,虚拟地址(即逻辑地址)由 16 位段选择子和 32 位段内偏移地址两部分组成。与实地址模式相比,段寄存器中存放的不是段基址,而是段选择子。段选择子的格式如图 14-3 所示。

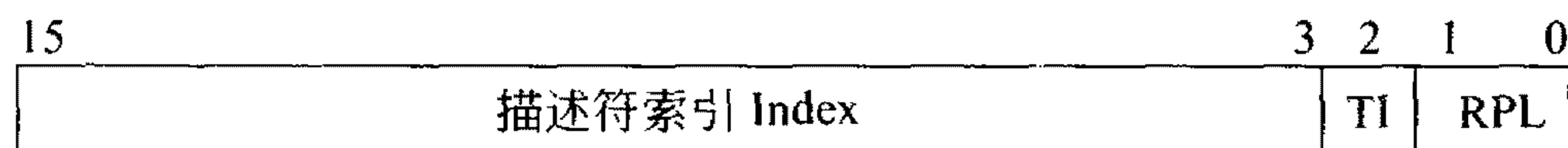


图 14-3 段选择子格式

Index: 描述符索引(Index)。描述符索引是指描述符在描述符表中的顺序号。Index 是 13 位,因此每个描述符表(GDT 或 LDT)最多有 $2^{13}=8192$ 个描述符。由于每个描述符长 8 字节,根据选择子的格式,屏蔽选择子低 3 位后所得的值就是选择子所指定的描述符在描述符表中的偏移。

TI: 表指示位(Table Indicator)。TI=0 指示从全局描述符表 GDT 中读取描述符; TI=1 指示从局部描述符表 LDT 中读取描述符。

RPL: 请求特权级(Requested Privilege Level),用于特权检查。

例: 假设某个选择子的内容是 0024H,则根据选择子的格式可知: Index=4, TI=1, RPL=0,所以它指定局部描述符表的第 4 个描述符,请求特权级为 0。

4. 虚拟空间的大小

一个任务除了可使用全系统各任务公用的全局描述符表 GDT 外,还有一个是任务自己的局部描述符表 LDT。因此使用的虚拟空间分为相等的两部分,一部分空间的描述符在全局描述符表中,另一部分空间的描述符在局部描述符表中。由于全局和局部描述符表都可以包含多达 8192 个描述符,而每个描述符所描述的段的最大值可达 4GB,因此最大的虚拟地址空间可为: $4GB \times 8192 \times 2 = 64TB$ 。

5. 全局描述符表寄存器 GDTR 和局部描述符表寄存器 LDTR

全局描述符表 GDT 的存储位置由全局描述符表寄存器 GDTR 指出。GDTR 长 48 位,其中高 32 位为基址,低 16 位为界限。

例如: 若 GDTR=0F002E0001FFH,则 GDT 的地址为 0F002E00,长度为 1FFH+1=

200H。GDT 中共有 $200\text{H}/8=40$ 个段描述符。

与 GDTR 不同,局部描述符表寄存器 LDTR 并不直接指出 LDT 的存储位置。LDTR 由程序员可见的 16 位寄存器和程序员不可见的 64 位高速缓冲寄存器组成。

由 LDTR 寄存器确定 LDT 位置的过程如图 14-4 所示。实际上,每个任务的局部描述符表 LDT 作为系统的一个特殊段,也由一个描述符描述。而这个 LDT 的描述符存放在 GDT 中。在初始化或任务切换过程中,把对应任务 LDT 的描述符的段选择子装入 LDTR 的 16 位寄存器,处理器根据装入 LDTR 可见部分的段选择子,从 GDT 中取出对应的描述符,并把 LDT 的基地址、段界限和属性等信息保存到 LDTR 的不可见的高速缓冲寄存器中。随后对 LDT 的访问,就可根据保存在高速缓冲寄存器中的有关信息进行。

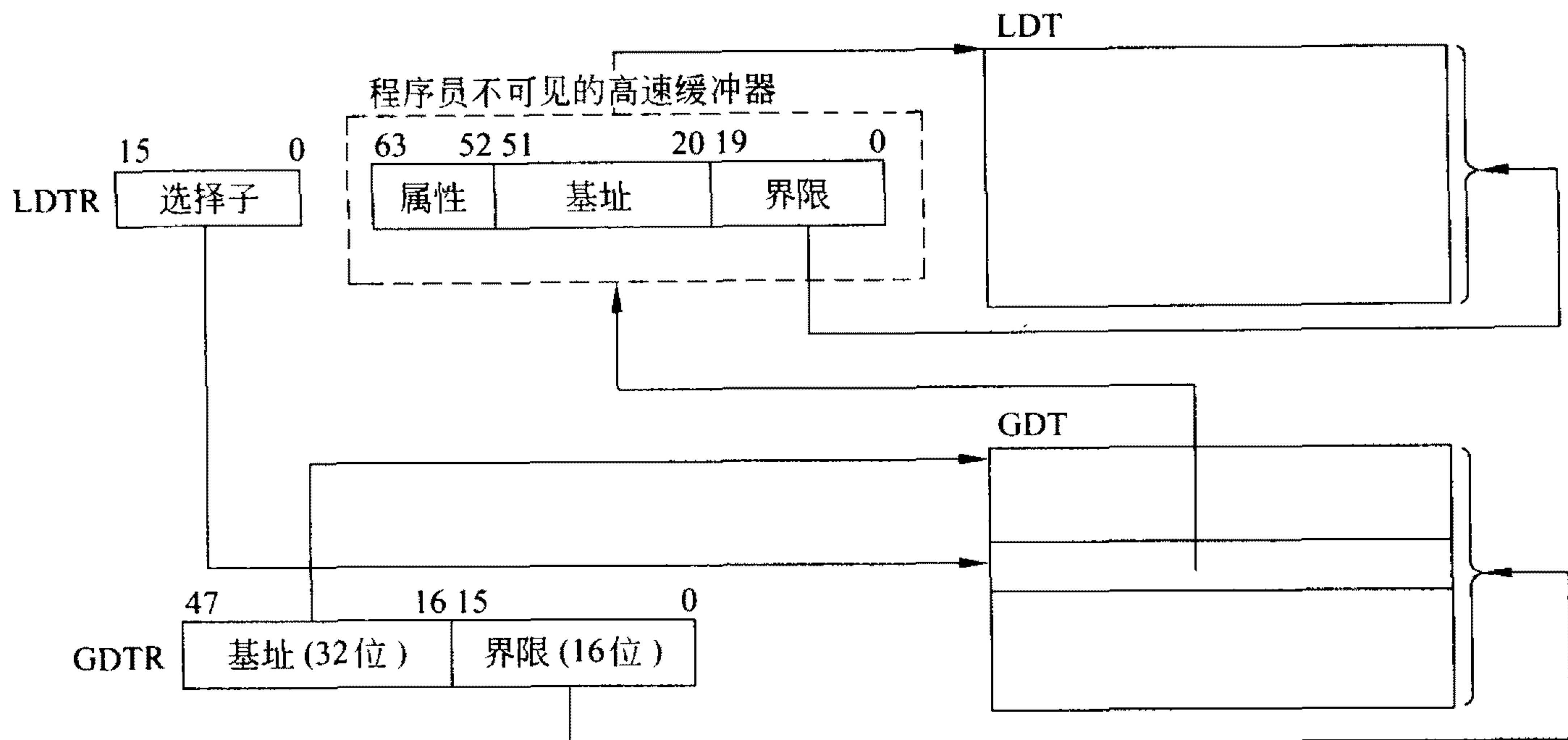


图 14-4 由 LDTR 确定 LDT 存储位置和界限

6. 虚拟地址到线性地址的转换

下面给出将虚拟地址转换为线性地址的过程,如图 14-5 所示。

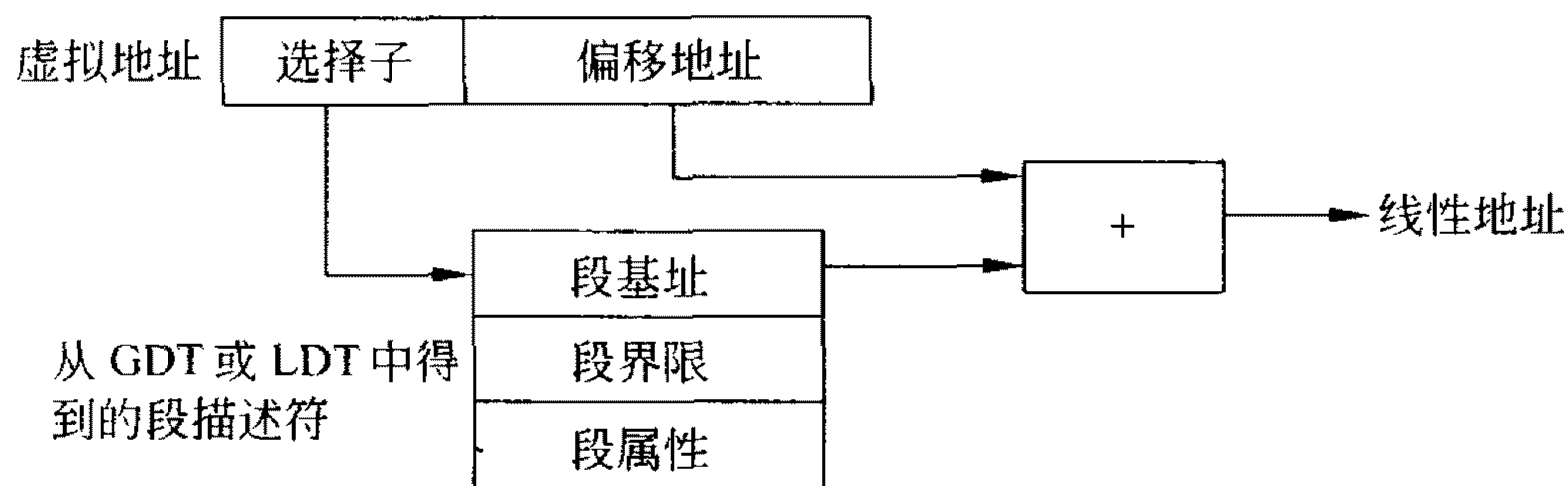


图 14-5 从虚拟地址转换到线性地址

① 首先,当操作系统把一个程序加载到内存时,会把对应的段选择子装入相应的段寄存器里;当一条指令去访问虚拟空间的某个地址时,系统首先会判断该地址属于哪个段,然后从相应的段寄存器中找到该段对应的选择子,先根据该选择子的 RPL 字段,进行特权检查,以判断该指令能否访问该地址;如果可以,再根据选择子的 TI 字段,判断该选择子是访问全局描述符表,还是局部描述符表。

② 如果该选择子是访问全局描述符表,那么先从全局描述符表寄存器 GDTR 中得到全局描述符表的基址;然后再用选择子的描述符索引 Index 字段作为索引,得到相应的描述符,从而得到相应段的段基址;这时再加上指令中的 32 位段内偏移地址就可以得到所对应的线性地址了。

③ 如果该选择子是访问局部描述符表,那么首先仍然是从全局描述符表寄存器 GDTR 中得到全局描述符表的基址,然后再用局部描述符表寄存器 LDTR 作为索引在全局描述符表中找到局部描述符表所在段的描述符,从而得到局部描述符表的基址,界限和属性(参见图 14-4);这时再利用选择子的描述符索引 Index 字段作为索引,在局部描述符表中得到相应段的段基址;这时再加上指令中的 32 位段内偏移地址,就可得到对应的线性地址了。

值得注意的是,从 80286 开始每个段寄存器都配有一个高速缓冲寄存器,称之为段描述符高速缓冲寄存器或描述符投影寄存器,对程序员而言它是不可见的。每当把一个选择子装入到某个段寄存器时,处理器都会自动从描述符表中取出相应的描述符,并把描述符中的信息保存到对应的高速缓冲寄存器中。此后对该段进行访问时,处理器都使用对应高速缓冲寄存器中的描述符信息,而不用再从描述符表中取描述符。从而提高了速度,改善了处理机性能。

14.1.2 分页管理

分页管理机制实现线性地址到物理地址的转换。分页管理机制是可选的,如果不启用分页管理机制,那么线性地址就是物理地址。

在保护模式下,控制寄存器 CR0 中的最高位 PG 位,控制分页管理机制是否生效。如果 PG=1,启动分页管理,把线性地址转换为物理地址。如果 PG=0,分页机制无效,线性地址就直接作为物理地址。分页管理只有在保护方式下才可以实现,即只有在 CR0 的 PE=1 的前提下,才能够使 PG=1。

1. 线性地址到物理地址的映射

分页管理机制将线性空间和物理空间分别划分为大小相同的块。每块称之为页。在大多数 32 位处理器中,页的大小固定为 4KB,页的边界是 4K 的倍数。因此,4GB 的地址空间被划分为 1 兆个页,页的开始地址具有 XXXXX000H 的形式。为此,将线性空间页开始地址的高 20 位 XXXXXH 称为虚页号,物理空间页开始地址的高 20 位 XXXXXH 称为实页号。虽然 32 位处理器在保护模式下,可寻址访问的物理空间和线性空间范围都为 4GB。但在实际情况下,物理存储器的容量要远小于 4GB 物理地址空间。因此,物理空间的实页号只能在实际存在的物理存储器的地址范围内选取。

由此可见,在把 32 位线性地址转换成 32 位物理地址的过程中,要解决的是线性空间的虚页号到物理空间实页号的映射,也就是线性地址高 20 位到物理地址高 20 位的转换。而映射过程中低 12 位页内偏移地址保持不变,也就是说,线性地址的低 12 位就是物理地址的低 12 位。

2. 页目录表、页表和页表项

线性空间的虚页号到物理空间的实页号之间的映射用映射表来描述。由于 4GB 的地址空间划分为 1 兆个页,因此,如果用一张映射表来描述这种映射,那么该表就要有 1 兆个表项,若每个表项占用 4 个字节,那么该表就要占用 4 兆字节。为避免占用如此巨大的存储器资源,所以大多数的 32 位处理器把映射表分为两级:页目录表和页表。

① 第一级称为页目录表,存储在一个 4KB 的物理页中。页目录表共有 1024 个表项,可以指定 1024 个页表。这些页表可以分散存放在任意的物理页中,而不需要连续存放。页目录表中每一个表项包含对应第二级页表所在物理空间页的实页号。

② 第二级称为页表,每张页表也安排在一个 4KB 的物理页中。每张页表都有 1024 个表项,可以指定 1024 个物理空间页,这些物理空间页也可任意地分散在物理地址空间中。每个表项包含对应物理空间页的实页号。需要注意的是,存储页目录表和页表的起始地址是 4K 的整数倍。

采用上述映射表结构,存储全部二级页表需要 4MB,此外还需要 4KB 用于存储页目录表。这样的两级映射表比单一的整张映射表多占用 4KB。实际上,不需要在内存中存储完整的两级映射表,对于线性地址空间中不存在的或未使用的部分不需要分配页表。除了必须给页目录表分配物理页外,只有在需要时,才给页表分配物理页,因此页映射表的大小对应于实际使用的线性地址空间大小。因为任何一个运行程序使用的线性地址空间远小于 4GB,所以用于分配给页表的物理页也远小于 4MB。

页目录表和页表中的每一个表项都为 4 字节长,都采用如图 14-6 所示的格式。

31	12	11	10	9	8	7	6	5	4	3	2	1	0
物理实页号 (20 位)				AVL	0	0	D	A	0	0	U/S	R/W	P

图 14-6 页目录表和页表的表项格式

物理实页号: 包含物理空间页的页号,也就是物理地址的高 20 位。

AVL: 由软件使用。

P: 存在位(Present)。P=1 表示表项有效,页表或页存在于物理内存中;P=0 表示表项无效。有效页表或页不在物理内存中。

D: 写入位(Dirty)。D=1 时表示对该页进行过写操作;D=0 时表示对该页还没有进行过写操作。它只用于页表的表项。

A: 访问标志(Accessed)。如果对某页表或页访问过,CPU 置 A 位为 1。

R/W 和 U/S: 用于对页进行保护。

3. 线性地址到物理地址的转换

图 14-7 说明了分页管理机制通过页目录表和页表实现 32 位线性地址到 32 位物理地址的转换过程。

① 从控制寄存器中 CR3 中得到页目录表所在物理页的页号。

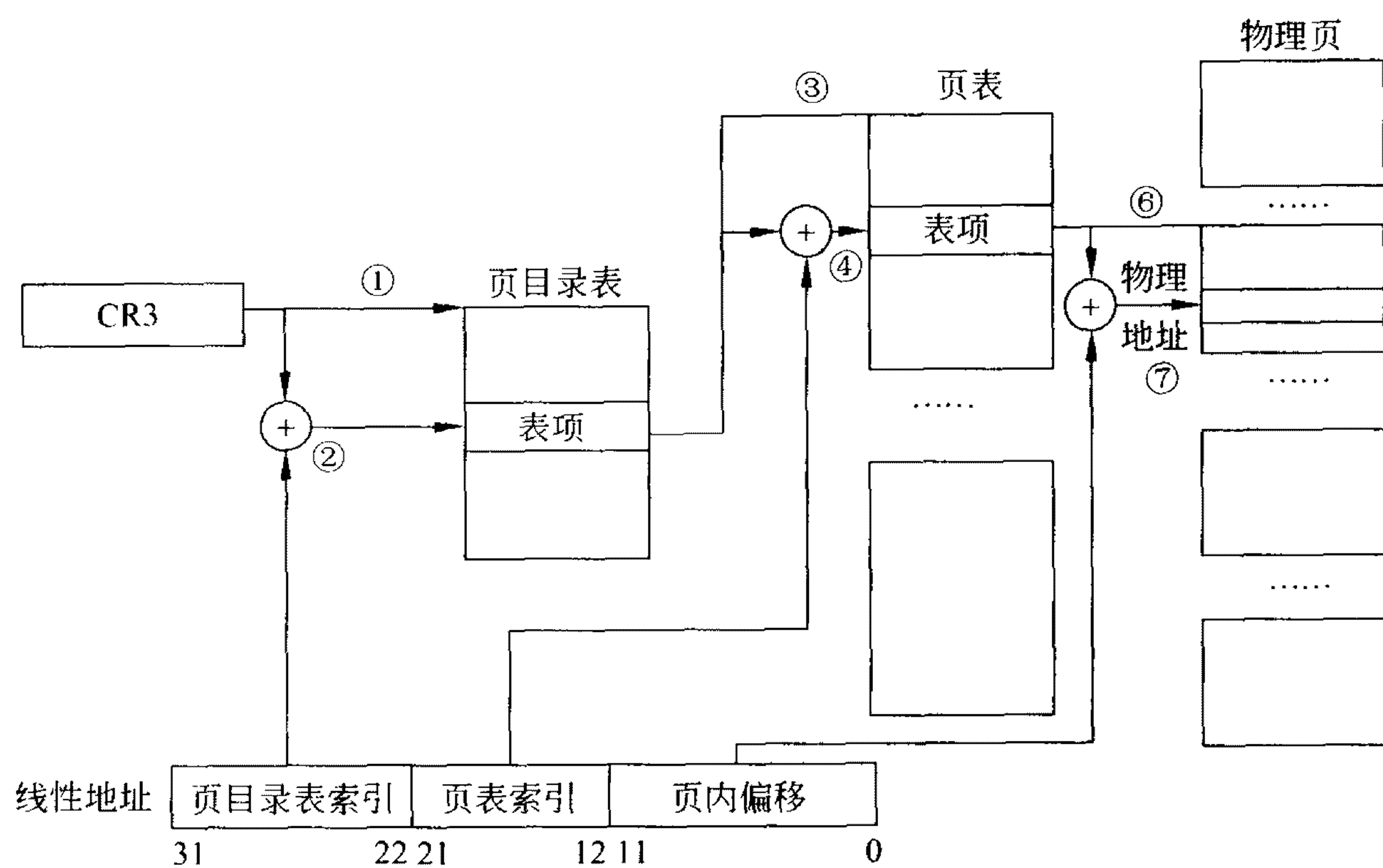


图 14-7 线性地址到物理地址的转换

② 将线性地址的最高 10 位(即位 22 至位 31)作为页目录表的索引,在页目录表找到对应表项,该表项的高 20 位给出了第二级页表所在物理页的实页号。

③ 然后将线性地址的中间 10 位(即位 12 至位 21)作为所指定页表的索引,在页表中找到对应表项,该表项的高 20 位给出了物理页的实页号。

④ 最后将物理页的实页号再作为高 20 位,把线性地址的低 12 位不加改变地作为 32 位物理地址的低 12 位,便得到 32 位线性地址对应的 32 位物理地址。

为了避免在每次存储器访问时都要访问内存中的页表,提高访问内存的速度,处理器的硬件把最近使用的线性-物理地址转换函数存储在处理器内部的页转换高速缓存中。在访问存储器页表之前总是先查阅高速缓存,仅当必须的转换不在高速缓存中时,才访问存储器中的两级页表。页转换高速缓存也称为页转换查找缓存,记为 TLB。

14.1.3 虚拟存储器

虚拟存储器是一项硬件和软件结合的技术。存储管理部件把物理存储器(主存)和辅存储器(磁盘)看作是一个整体,即虚拟存储器。虚拟存储器的容量可以达到 64TB,程序员可以在这个地址范围内编程,编写程序的大小可以超过实际配置的物理存储器的容量。在程序运行的任何时刻,都只是把虚拟地址空间的一小部分映射到主存,其余部分则存储在磁盘上。当程序所访问的页(段)不在物理存储器时,操作系统再把那一部分从辅存调入主存。

32 位处理器的分段、分页存储管理机制从硬件上很好地支持了虚拟存储器的实现。

① 存储段描述符和页目录表、页表的表项中都设置了存在位 P。当 P=1 时,表示存储段描述符指定的段或表项指定的页,存在于物理存储器中;当 P=0,表示存储段描述符指定的段或表项指定的页不在物理存储器中。如果程序访问不存在的段(页),会引起异

常,这时操作系统可以把该不存在的段(页)从磁盘读入,把 P 位置为 1,然后使引起异常的程
序恢复运行。

② 存储段描述符和页目录表、页表的表项中都设置了访问位 A。A=1 表示相应段或物理页在最近一段时间被访问过。通过周期性地检测及清除 A 位,操作系统就可确定哪些(段)页在最近一段时间未被访问过。当物理存储器资源紧缺时,这些最近未被访问的(段)页很可能就被选择出来,将它们从物理存储器换出到磁盘上去。

14.1.4 保护机制

为了支持多任务,从 80286 开始,处理器就具备了保护机制。

1. 不同任务之间的保护和共享

通过把每个任务放置在不同的虚拟地址空间的方法,来实现任务与任务的隔离,达到应用程序之间保护的目
的。每个任务都有自己的局部描述符表,随着任务的切换,系统当前的局部描述符表也随之切换,使各个任务私有的各个段与其他任务相隔离。每个任务还有自己的页目录表和页表,这样即使多个任务中的代码位于相同的线性地址,通过各自的页表,也会把这些相同的线性地址映射到不同的物理地址。从而使各任务空间得以隔离,实现任务间的保护。

在系统中只有一个全局描述符表,它含有每一个任务都可能或可以访问的段的描述符。在任务切换时,全局描述符表并不切换。由于每个任务都是访问同一个全局描述符表,因此通过全局描述符表可以使各任务都需要使用的段能够被共享。

分页机制也对共享提供了支持。每一个任务可使用自己的页映射表独立地实现线性地址到物理地址的转换。但是,如果使每一个任务所用的页映射表具有部分相同的映射,那么也就可以实现部分页的共享。

2. 同一任务内的段级保护

在一个任务之内,定义有四种执行特权级,用于限制对任务中的段进行访问。特权级用 PL 表示,分别为特权 0,1,2,3 级。0 级的特权级最高,1 级次之,3 级的特权级最低。在段级保护中使用了三种形式的特权管理,当前特权级(CPL)、描述符特权级(DPL)和请求特权级(RPL)。

当前特权级(CPL):在任何时候,一个任务总是在四个特权级之一下运行,当前运行程序的特权级称为当前特权级。CPL 存放在 CS 寄存器的 RPL 字段内,每当一个代码段选择子装入 CS 寄存器中时,处理器自动地把 CPL 存放到 CS 的 RPL 字段。

描述符特权级(DPL):描述符特权级是由段描述符中的 DPL 确定。它规定了访问该描述符所描述的段的任务的最低级别。只有当 CPL 级别等于或高于 DPL 时,当前任务才能访问该段。

请求特权级(RPL):段选择子特权级是由段寄存器中段选择子的 RPL 确定。使用段选择子的 RPL 字段,将改变特权级的测试规则。在这种情况下,与所访问段的特权级相比较的特权级不是 CPL,而是 CPL 与 RPL 中级别低的特权级。

通常把操作系统的核心部分放在 0 级,操作系统的其余部分放在 1 级,而应用程序放在 3 级,留下的 2 级供中间软件使用。这样在 0 级的操作系统核心有权访问任务中的所有数据段,而在 3 级的应用程序只能访问程序本身的,也是在 3 级的数据段,不能访问同级的其他应用程序的数据段。

位于某特权级的程序,只允许访问同一级别或低级别的数据;而位于某特权级的程序只允许转移到同一任务内同一级别或更高级别的程序代码(在任务间转移允许转移到任何特权级)。这些保护措施使得保护模式下的操作系统能够安全稳定地运行。

3. 同一任务内的页级保护

分页机制只区分两种特权级。特权级 0、1 和 2 统称为系统特权级,特权级 3 称为用户特权级。页目录表和页表的表项中的保护属性位 R/W 和 U/S 就是用于对页进行保护。

R/W: 读写属性位,指示该表项所指定的页是否可读、写或执行。若 R/W=1,则对表项所指定的页可进行读、写或执行;若 R/W=0,则对表项所指定的页可读或执行,但不能对该指定的页写入。但是,R/W 位对页的写保护只在处理器处于用户特权级时发挥作用;当处理器处于系统特权级时,R/W 位被忽略,即总可以读、写或执行。

U/S: 用户/系统属性位,指示该表项所指定的页是否是用户级页。若 U/S=1,则表项所指定的页是用户级页,可由任何特权级下执行的程序访问;若 U/S=0,则表项所指定的页是系统级页,只能由系统特权级下执行的程序访问。表 14-2 列出了在上述属性位 R/W 和 U/S 所确定的页级保护下,用户级程序和系统级程序分别具有的对用户级页和系统级页进行操作的权限。

表 14-2 U/S 和 R/W 的保护作用

页级保护属性	U/S	R/W	用户级访问权限	系统级访问权限
	0	0	无	读/写/执行
	0	1	无	读/写/执行
	1	0	读/执行	读/写/执行
	1	1	读/写/执行	读/写/执行

页目录表项中的保护属性位 R/W 和 U/S,对由该表项指定页表的全部 1024 个页起到保护作用。所以当页目录表表项中的 U/S 和 R/W 与页表表项中的 U/S 和 R/W 不一致时,按两者共同允许的功能执行。和分页机制是在分段机制之后起作用一样,页级保护也在段级保护之后起作用。先测试有关的段级保护,如果启用分页机制,那么在检查通过后,再测试页级保护。如果两者功能不一致时,也按两者共同允许的功能执行。如段的类型为读/写,而页规定为只允许读/执行,那么不允许写;如果段的类型为只读/执行,那么不论页保护如何,也不允许写。

14.2 保护模式下的程序调用和转移

本书在第 3 章指令系统中,已经对实地址模式下的转移和调用指令 JMP,CALL 及 RET 作了介绍。保护模式下的调用和转移可分为两大类:

- ① 同一任务内的调用和转移。
- ② 任务间的调用和转移(任务切换)。

同一任务内的调用和转移又可分为:段内调用和转移、段间调用和转移。段内调用和转移的过程与实地址模式下相似,仅仅是改变指令指针 EIP 的内容,而不涉及特权级变换和任务切换。因此本节对保护模式下任务内的段间调用和转移以及任务切换进行介绍。首先给出几个相关的几个基本概念。为了方便,这里将转移和调用统称为转移。

14.2.1 系统段描述符、门描述符和任务状态段

保护模式支持多任务,每个任务都有自己的局部描述符表 LDT。另外每个任务还有一个任务状态段 TSS,用于保存任务的有关信息。LDT 和 TSS 作为系统的一个特殊段,由系统段描述符描述,描述符存放在 GDT 中。

1. 系统段描述符

系统段描述符以及属性的格式如图 14-8 所示。

7			0
段界限 (位 7~0)			0
段界限 (位 15~8)			1
段基址 (位 7~0)			2
段基址 (位 15~8)			3
段基址 (位 23~16)			4
段属性			5
段属性	段界限 (位 19~16)		6
段基址 (位 31~24)			7

(a)

位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0	第 5 字节
P	DPL		DT=0	TYPE				

位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0	第 6 字节
G		0	AVL	段界限 (位 19~16)				

(b)

图 14-8 系统段描述符及属性的格式

可以看出,系统段描述符的格式与图 14-1 存储段描述符的格式相似,区分的标志是属性字节中的描述符类型位 DT 的值。DT=1 表示存储段,DT=0 表示系统段。系统段描述符中的段基址和段界限字段与存储段描述符中的意义完全相同;属性中的 G 位、AVL 位、P 位和 DPL 字段的作用也完全相同。存储段描述符属性中的 D 位在系统段描述符中不使用。系统段描述符的类型字段 TYPE 编码及表示的类型如表 14-3 所示,其含义与存储段描述符的类型完全不同。

表 14-3 系统段描述符中 TYPE 定义

TYPE	说明	TYPE	说明
0000	未定义	1000	未定义
0001	可用 286TSS	1001	可用 386TSS
0010	LDT	1010	未定义
0011	忙的 286TSS	1011	忙的 386TSS
0100	286 调用门	1100	386 调用门
0101	任务门	1101	未定义
0110	286 中断门	1110	386 中断门
0111	286 陷阱门	1111	386 陷阱门

从表 14-3 可见,只有类型编码为 0001、0010、0011、1001 和 1011 的描述符才是真正的系统段描述符,它们用于描述系统段 LDT 和任务状态段 TSS,其他类型的描述符是门描述符。

2. 门描述符

门描述符的格式及属性格式如图 14-9 所示,其中 P,DPL,DT,TYPE 的定义和系统段描述符相同,双字计数(Dword Count)字段是要传递到被调用过程的双字参数的数量。其他字节主要用于存放一个 48 位的全指针(16 位的选择子和 32 位的偏移地址)。

门描述符并不描述某种内存段,而是描述控制转移的入口点。这种描述符就像一个通向另一代码段的门。通过这种门,可实现任务内特权级的变换和任务间的切换。门描述符又可分为:任务门、调用门、中断门和陷阱门。调用门描述某个子程序的入口,通过调用门可实现任务内从低特权级变换到高特权级;任务门指示任务,通过任务门可实现任务间的切换;中断门和陷阱门描述中断/异常处理程序的入口点。双字计数字段只对调用门有效,而对中断门、陷阱门和任务门而言无意义。

3. 任务状态段

任务状态段(Task State Segment)是保存一个任务重要信息的特殊段。任务状态段描述符属于系统段描述符,格式见图 14-8。当前任务的任务状态段可由任务状态段寄存器 TR 寻址。和局部描述符表寄存器 LDTR 相同,任务状态段寄存器 TR 也有程序员可见和不可见两部分。任务状态段寄存器 TR 的可见部分含有当前任务状态段描述符的选择子,TR 的不可见的高速缓冲寄存器部分含有当前任务状态段的段基址和段界限等信息。

7		0
偏移地址 (位 7~0)		0
偏移地址 (位 15~8)		1
选择子 (位 7~0)		2
选择子 (位 15~8)		3
门属性	双字计数	4
门属性		5
偏移地址 (位 23~16)		6
偏移地址 (位 31~24)		7

(a)

位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0	
0	0	0	双字计数					第 4 字节
位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0	
P	DPL	DT=0	TYPE					第 5 字节

(b)

图 14-9 门描述符的格式及属性

TSS 在任务切换过程中起着重要的作用,通过它实现任务的挂起和恢复。任务切换是指,挂起当前正在执行的任务,恢复或启动另一任务的执行。当任务被挂起时,当前处理器中各寄存器的值被自动保存到 TR 所指定的 TSS 中;当任务恢复时,把保存在 TSS 中的各寄存器的值送到处理器的各寄存器中,使任务继续运行。

任务状态段 TSS 的基本格式如图 14-10 所示。

从图中可见,TSS 的基本格式由 104 个字节组成。这 104 个字节的基本格式是不可改变的,但在此之外还可定义若干附加信息。基本的 104 个字节可分为链接字段、内层堆栈指针、地址映射寄存器、寄存器保存和其他字段等五个区域。

(1) 链接字段

链接字段安排在 TSS 内从偏移 0 开始的双字中,其高 16 位未用。低 16 位保存前一个被挂起的任务的 TSS 描述符的选择子。如果当前的任务被段间调用指令 CALL 或中断/异常激活,那么链接字段保存被挂起的任务的 TSS 的选择子,并且标志寄存器 EFLAGS 中的 NT 位被置 1,使链接字段有效。在返回时,由于 NT 标志位为 1,中断返回指令 IRET 沿链接字段恢复到链上的前一个任务。

(2) 内层堆栈指针区域

为了有效地实现保护,同一个任务在不同的特权级下,使用不同的堆栈。当从某一个特权级 A 变换到另一特权级 B 时,任务使用的堆栈也同时从 A 级变换到 B 级。

TSS 的内层堆栈指针区域中有三个堆栈指针,它们都是 48 位的全指针(16 位的选择子 SS 和 32 位的偏移 ESP),分别指向 0 级、1 级和 2 级堆栈的栈顶。当发生向高特权级转移时,把该特权级堆栈指针装入 SS 及 ESP 寄存器以变换到高特权级堆栈。并将低特

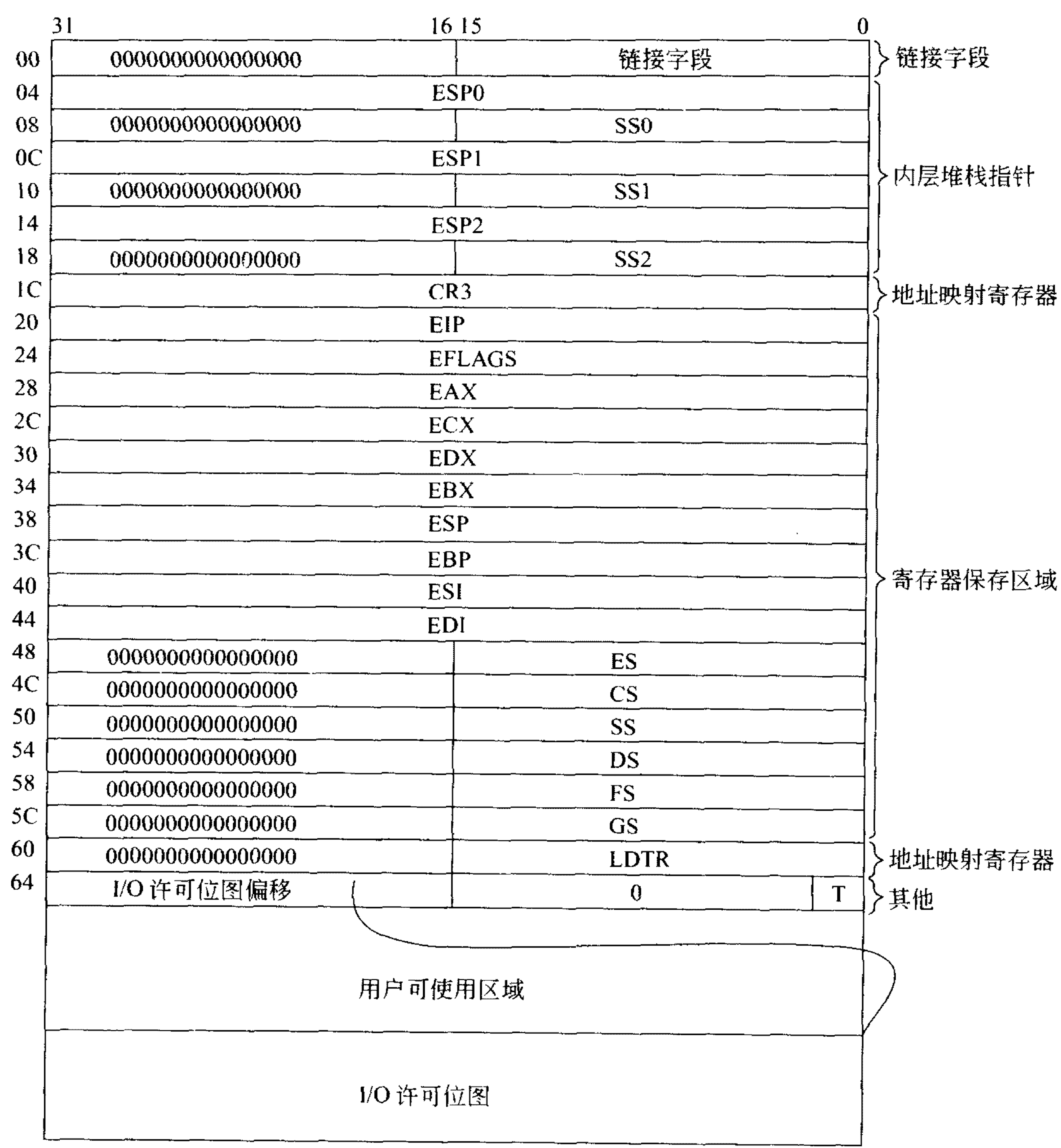


图 14-10 TSS 的格式

权级堆栈的指针压栈,保存在高特权级堆栈中。没有指向 3 级堆栈的指针,因为 3 级是最低特权级,所以任何一个向高特权级的转移都不可能转移到 3 级。

(3) 地址映射寄存器区域

为了实现任务间的保护,每个任务都有自己的局部描述符表 LDT 和页目录表,而 LDT 由 LDTR 确定,页目录表起始的物理地址由控制寄存器 CR3 确定。所以,随着任务的切换,处理器自动从要执行任务的 TSS 中取出 LDTR 和 CR3 这两个字段,分别装入到寄存器 CR3 和 LDTR。这样就改变了虚拟地址空间到物理地址空间的映射。但是,在任务切换时,处理器并不会自动把换出任务的寄存器的 CR3 和 LDTR 的内容保存到 TSS 中的地址映射寄存器区域,这需要由程序完成。

(4) 寄存器保存区域

寄存器保存区域,用于保存通用寄存器、段寄存器、指令指针和标志寄存器。当 TSS

对应的任务正在执行时,保存区域是未定义的;在当前任务被切换出时,这些寄存器的当前值就保存在该区域。当下次切换回原任务时,再从保存区域恢复出这些寄存器的值,从而使处理器恢复成该任务换出前的状态,最终使任务能够恢复执行。

(5) 其他字段

为了实现输入/输出保护,要使用 I/O 许可位图。任务使用的 I/O 许可位图也存放在 TSS 中,作为 TSS 的扩展部分。在 TSS 内偏移 66H 处的字用于存放 I/O 许可位图在 TSS 内的偏移(从 TSS 开头开始计算)。关于 I/O 许可位图的作用,在 14.3 节中将会详细介绍。TSS 中的 T 位为调试陷阱位。在发生任务切换时,如果进入任务的 T 位为 1,那么在任务切换完成之后,新任务的第一条指令执行之前产生调试陷阱。

14.2.2 任务内的段间转移

1. 任务内无特权级变换的转移

与实地址模式下相似,JMP 和 CALL 也可分为段间直接转移和段间间接转移两类。如果指令 JMP 和 CALL 在指令中直接给出目标地址,那么就是段间直接转移。指令格式如“JMP XX: YY”、“CALL XX: YY”。其中 XX 是 16 位代码段选择子,YY 是偏移地址。在 32 位代码段中,偏移地址用 32 位表示,在 16 位代码段中,偏移地址只使用 16 位表示。

处理器在执行段间直接转移指令时,首先通过段选择子从全局描述符表或局部描述符表中取得目标代码段描述符,装载到 CS 高速缓冲寄存器;然后将段选择子装入 CS 段寄存器,偏移地址装入指令指针寄存器 EIP,CPL 存入 CS 内选择子的 RPL 字段;如果是执行 CALL 指令,还需将返回地址指针压栈,从而完成向目标代码段的转移。上述步骤只是对转移过程的简单说明。

在将目标代码段描述符内的有关内容装载到 CS 高速缓冲寄存器时,处理器要进行如下的特权级检测:对于普通代码段,要求 CPL 等于 DPL,RPL 的级别高于或等于 DPL。

由此可见,在直接转移的情况下,如果目标代码段是普通代码段,只能转移到特权级相同的代码段。因此利用段间直接转移指令 JMP 或调用指令 CALL 可进行任务内无特权级变换的转移。在通常情况下,RET 指令与 CALL 指令相对应。在利用 CALL 指令以任务内无特权级切换的方式转移到某个子程序后,在子程序内可利用 RET 指令以任务内无特权级切换的方式返回主程序。

2. 任务内特权级变换的转移

利用段间直接转移指令 JMP 或调用指令 CALL 只能进行任务内无特权级变换的转移。而在实际应用中,位于低特权级的应用程序往往需要调用高特权级的操作系统程序来完成一些功能,如打开、关闭、读写文件,申请一块物理内存等。这种使控制权从较低的特权级转移到高特权级(即发生了任务内特权级变化)的转移,需要利用间接转移的方法来实现。

如果指令 JMP 和 CALL 在指令中给出的是包含目标地址的门描述符选择子,那么就是段间间接转移。指令格式如“JMP XX: YY”、“CALL XX: YY”。其中 XX 不再是 16 位代码段选择子,而是一个门选择子;偏移地址 YY 没有使用。

在同一任务内,实现特权级从低到高变换的方法是利用 CALL 指令,通过调用门进行转移;实现特权级从高到低变换的方法是利用 RET 指令。注意,JMP 指令只能实现无特权级变换的转移,不能实现任务内不同特权级的变换。

CALL 指令通过调用门转移的过程如图 14-11 所示。①根据指令中的调用门选择子,从描述符表得到一个调用门描述符;②根据门选择符中的 16 位选择子来读取目标代码段描述符,目标代码段描述符给出了被调用段的段基址;③使用门描述符中的偏移地址代替 CALL 指令中的偏移地址来控制目标代码段的入口点。上述步骤只是对转移过程的简单说明。

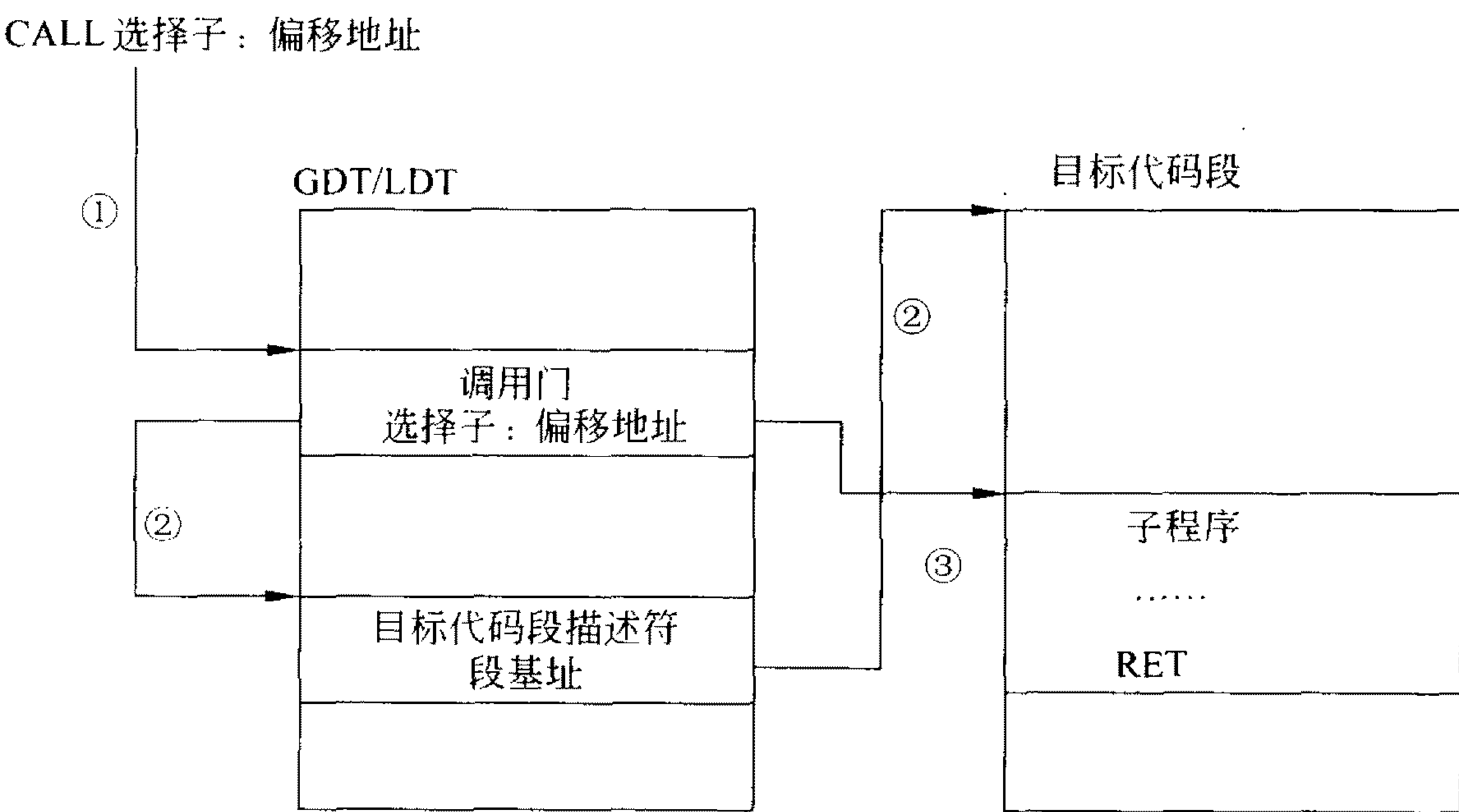


图 14-11 使用调用门的转移过程

在使用调用门进行段间转移时,CPU 也要进行特权级检测,检查当前任务特权级 CPL、门选择子请求特权级 RPL,门描述符的特权级为 DPL_GATE,目标代码段描述符特权级 DPL。只有当以下几个条件都满足时,才允许使用调用门:

- ① CPL 高于或等于 DPL_GATE;
- ② RPL 高于或等于 DPL_GATE;
- ③ CPL 低于或等于 DPL。

对于普通代码段,当 CPL=DPL 时,仍发生无特权级变换的转移;当 CPL 低于 DPL 时,就发生向高特权级变换的转移,将调用门中的选择子和偏移装入 CS 和指令指针 EIP 中,并使 CPL 等于 DPL,同时切换到高特权级堆栈。由此可见,使用段间调用指令 CALL,通过调用门可以实现从低特权级程序调用进入高特权级程序;通过调用门也可实现无特权级变换的转移。需要注意的是,JMP 指令和 CALL 指令都不能实现向低特权级的转移,否则会引起异常。

在使用 CALL 指令通过调用门向高特权级转移时,不仅特权级发生变换,而且也切换到高特权级的堆栈段。根据变换后的特权级,使用 TSS 中相应的堆栈指针对 SS 及

ESP 寄存器进行初始化,建立起一个空栈。然后处理器先将低特权级堆栈的指针 SS 及 ESP 寄存器的值压入高特权级堆栈,再将 CALL 指令需要传递的参数压入堆栈,压入参数的数量由调用门中的 DCOUNT 字段给出。最后,将调用的返回地址 CS 和 EIP 压入堆栈,以便在调用结束时返回。

在使用 RET 指令返回时,首先把低特权级的 CS 和 EIP 弹出堆栈,然后调整 ESP 指针跳过传递的参数,最后把低特权级的 SS 和 ESP 弹出,以恢复低特权级堆栈。当然上述过程也需要进行一系列的安全检查。

14.2.3 任务间的转移

利用段间转移指令 JMP 或者段间调用指令 CALL,通过任务门或直接通过任务状态段,可以进行任务间的转移,即任务切换。此外,在中断/异常或者执行 IRET 指令时也可能发生任务切换。因为 RET 指令的目标地址只能使用代码段描述符,所以,不能通过 RET 指令实现任务切换。

1. 直接通过 TSS 的任务切换

当段间转移指令 JMP XX:YY 或段间调用指令 CALL XX:YY 中的选择子指向一个可用任务状态段 TSS 描述符时,正常情况下就发生从当前任务到可用 TSS 对应任务(目标任务)的直接切换。目标任务的人口点由目标任务 TSS 内的 CS 和 EIP 字段所规定的指针确定。

2. 间接通过任务门的任务切换

当段间转移指令 JMP XX:YY 或段间调用指令 CALL XX:YY 中的选择子指向一个任务门时,正常情况下就发生任务切换,即从当前任务切换到由任务门内的选择子所指向的 TSS 相应任务(目标任务)。

用 JMP/CALL 指令进行直接/间接任务转换时,要对 TSS/任务门进行保护。即只有相同级别或特权级更高的程序可以访问它。如果通过了 TSS/任务门特权级检查,就可进入目标任务的任一特权级。

根据目标任务 TSS 描述符进行任务切换的主要过程如下:

- ① 把 CPU 各寄存器的值保存到当前任务的 TSS。
- ② 把指向目标任务 TSS 的选择子装入 TR 寄存器中。同时把对应 TSS 的描述符装入 TR 的高速缓冲寄存器中。此后,当前任务改称为原任务,目标任务改称为当前任务。
- ③ 根据保存在当前任务(目标任务)TSS 中的内容,恢复各寄存器,包括 CR3 寄存器的值。
- ④ 进行链接处理。如果需要链接,那么将指向原任务 TSS 的选择子写入当前任务 TSS 的链接字段,把当前任务 TSS 描述符类型改为“忙”位,并将标志寄存器 EFLAGS 中的 NT 位置 1,表示是嵌套任务。由 JMP 指令引起的任务切换不实施链接/解链处理;由 CALL 指令、中断、IRET 指令引起的任务切换要实施链接/解链处理。
- ⑤ 把 TSS 中的 CS 选择子的 RPL 作为当前任务特权级设置为 CPL。任务切换可以

在一个任务的任何特权级发生,并且可以切换到另一个任务的任何特权级。

⑥ 装载 LDTR 寄存器,代码段寄存器 CS、堆栈段寄存器 SS 和各数据段寄存器及其高速缓冲寄存器。堆栈段使用的是 TSS 中的 SS 和 SP 字段的值,即使发生了向高特权级的变换也不使用高特权级栈保存区中的指针。这与任务内通过调用门的转移不同。

一个任务有忙和可用两种状态,由 TSS 描述符中的类型 TYPE 表示。如果一个任务是当前正在执行的任务或是被 TSS 中链接字段挂起并链接的任务,则该任务状态为忙。否则就是一个可用任务。CALL、JMP 和中断都必须转移到一个可用任务。JMP 引起任务切换时,不实施链接,不导致任务的嵌套。在段间调用指令 CALL 引起任务切换时,实施链接,在切换过程中把目标任务置为“忙”,原任务仍保持“忙”;标志寄存器 EFLAGS 中的 NT 位被置为 1,表示任务是嵌套任务。

14.3 保护模式下的中断和异常

8086/8088 将中断分为内部中断和外部中断。内部中断又可分为 CPU 中断(如 0 型除法错中断)和软件中断(如 INT 21H)。外部中断是由 CPU 以外的器件发出的中断请求信号而引发的中断,又可称为硬件中断。为了支持多任务和虚拟存储器等功能,80386 及以后的处理器在保护模式下,将外部中断(硬件中断)称为“中断”,而把内部中断称为“异常”。CPU 最多处理 256 种中断或异常,每种中断或异常都分配了一个 0~255 的中断号(又称中断类型码)。

14.3.1 中断和异常分类

在保护模式下,中断仍然和实地址模式下相同,可分为可屏蔽中断和非屏蔽中断。非屏蔽中断所对应的中断号固定为 2,可屏蔽中断所对应的中断号由系统 8259 芯片提供。

异常是 80386 在执行指令期间检测到不正常的或非法的条件而引起的。软中断指令 INT n 也归类于异常。异常发生后,处理器就像响应中断那样处理异常。即根据中断号,转到相应的中断处理程序。

根据引起异常的程序是否可被恢复和恢复点不同,异常可分为故障(Fault)、陷阱(Trap)和中止(Abort)。对应的异常处理程序分别称为故障处理程序、陷阱处理程序和中止处理程序。

① 故障是在引起异常的指令之前,把异常情况通知给系统的一种异常。故障是可排除的。当控制转移到故障处理程序时,保存指向引起故障指令的 CS 及 EIP 值。这样,在故障处理程序把故障排除后,执行 IRET 返回到引起故障的程序继续执行时,刚才引起故障的指令就可以重新得到执行。这种重新执行,不需要操作系统软件的额外参与。

故障的发现可能在指令开始执行之前,也可能在指令执行期间。如果在指令执行期间检测到故障,那么中止故障指令,并把指令的操作数恢复为指令开始执行之前的值。这可以保证故障指令的重新执行得到正确的结果。例如,在一条指令的执行期间,如果

发现段不存在,那么停止该指令的执行,并通知系统产生段故障,对应的段故障处理程序可以通过加载该段的方法来排除故障,这样,原指令就可以成功执行,至少不再发生段不存在的故障。

② 陷阱是在引起异常的指令之后,把异常情况通知给系统的一种异常。当控制转移到异常处理程序时,保存指向引起陷阱的指令下一条要执行的指令的断点的 CS 及 EIP 的值。在转入陷阱处理程序时,引起陷阱的指令应正常完成。如软中断指令、单步异常都是陷阱异常。

③ 中止是在系统出现严重情况时,通知系统的一种异常。引起中止的指令是无法确定的。产生中止时,正执行的程序不能被恢复执行,系统可能要重新启动操作系统才能恢复正常运行状态。硬件故障或系统表中出现非法值或不一致的值是中止的例子。

14.3.2 中断和异常类型

CPU 能识别多种不同类别的中断和异常并赋予对应的中断号,如表 14-4 所示。某些异常还以出错码的形式提供一些信息传递给异常处理程序,出错代码列中的“无”表示没有出错代码,“有”表示有出错代码。

表 14-4 中断和异常一览表

中断号	中断和异常名称	中断和异常类型	出错代码	相关指令
0	除法出错	故障	无	DIV/IDIV
1	调试异常	故障/陷阱	无	任何指令
2	NMI	中断	无	
3	单字节 INT3	陷阱	无	INT 3
4	溢出	陷阱	无	INTO
5	边界检查	故障	无	BOUND
6	非法操作码	故障	无	非法指令编码或操作数
7	设备不可用	故障	无	浮点指令或 WAIT
8	双重故障	中止	有	任何指令
9	协处理器段越界	中止	无	访问存储器的浮点指令
0AH	无效 TSS 异常	故障	有	JMP、CALL、IRET 或中断
0BH	段不存在	故障	有	装载段寄存器的指令
0CH	堆栈段异常	故障	有	装载 SS 段的指令
0DH	通用保护异常	故障	有	任何特权指令、任何访问存储器的指令
0EH	页异常	故障	有	任何访问存储器的指令
10H	协处理器出错	故障	无	浮点指令或 WAIT
11H-0FFH	软中断	陷阱	无	INTn
	硬件中断	中断		

由表 14-4 可见,保护模式下,某些异常的中断号分配与实地址模式的可屏蔽中断号发生了冲突。实地址模式下的中断号的分配基于 PC 微型计算机系统的 8086/8088 CPU,上表中的中断号的分配是 80386 所规定的。因此在保护模式下必须重新设置

8259A 中断控制器,将可屏蔽硬件中断的中断号设置在 20H~FFH 之间,以避免异常相互冲突。例如,将实时时钟中断号设置为 38H,将键盘中断号设置为 31H。

14.3.3 中断和异常的处理过程

在实地址模式下,CPU 响应中断后,根据中断号从中断向量表中取得相应的中断向量(即中断服务子程序的入口地址),从而去执行中断服务程序。而在保护模式下,CPU 是根据中断号从中断描述表 IDT 中取得相应的门描述符,从而获得中断或异常处理程序的入口地址。

1. 中断描述符表 IDT

和全局描述符表 GDT 相同,整个系统只有一个中断描述符表 IDT。由于 CPU 最多处理 256 种中断或异常,而门描述符是 8 个字节长,所以 IDT 最大长度是 2KB。中断描述符表寄存器 IDTR 指示 IDT 在内存中的位置。和 GDTR 一样,IDTR 也是 48 位的寄存器,其中高 32 位为基址,低 16 位为界限。

中断描述符表 IDT 所含的描述符只能是中断门、陷阱门和任务门。也就是说,在保护模式下,CPU 只有通过中断门、陷阱门或任务门才能转移到对应的中断或异常处理程序。由于门描述符是 8 个字节长,因此中断或异常产生时,CPU 以中断号乘 8 从 IDT 中取得对应的门描述符,分解出选择子、偏移量和描述符属性类型,并进行有关检查。最后,根据门描述符的类型是中断门、陷阱门还是任务门,分情况转入中断或异常处理程序。

2. 通过中断门或陷阱门的中断/异常处理过程

如果中断号指示的门描述符是 386 中断门或 386 陷阱门,那么控制转移到当前任务的一个处理程序,并且可以变换特权级。与其他调用门的 CALL 指令一样,从中断门和陷阱门中获取指向处理程序的 48 位全指针。其中 32 位偏移地址送给 EIP,16 位选择子是对应处理程序代码段的选择子,它被送给 CS 寄存器,并根据选择子中的 TI 位是 0 或 1,从全局描述符表 GDT 或局部描述符表 LDT 中取得代码段描述符;这时,代码段描述符中的基地址确定了处理程序的段基址,EIP 确定了处理程序的入口地址,CPU 转向执行处理程序。整个过程如图 14-12 所示。

在上述的中断/异常处理过程中,CPU 会进行一系列的保护检测。如中断门或陷阱门中的选择子必须指向描述一个可执行代码段的描述符。如果选择子为空,就会引起通用保护故障,出错码是 0。另外还需注意以下几点:

① 中断或异常可以转移到同一特权级或高特权级处理程序。处理程序代码段的描述符中的类型及 DPL 字段,决定了这种同一任务内的转移是否要发生特权级变换。因此,在使用中断门时,中断处理程序通常必须被安排在特权级 0。

② 和在实地址模式下相同,CPU 在转向执行处理程序之前,首先要将标志寄存器和断点(EFLAGS,CS,EIP)压栈,以便中断返回。但这里每一次压栈操作是一个双字,CS 被扩展成 32 位。另外若有异常,还把出错码压入堆栈。

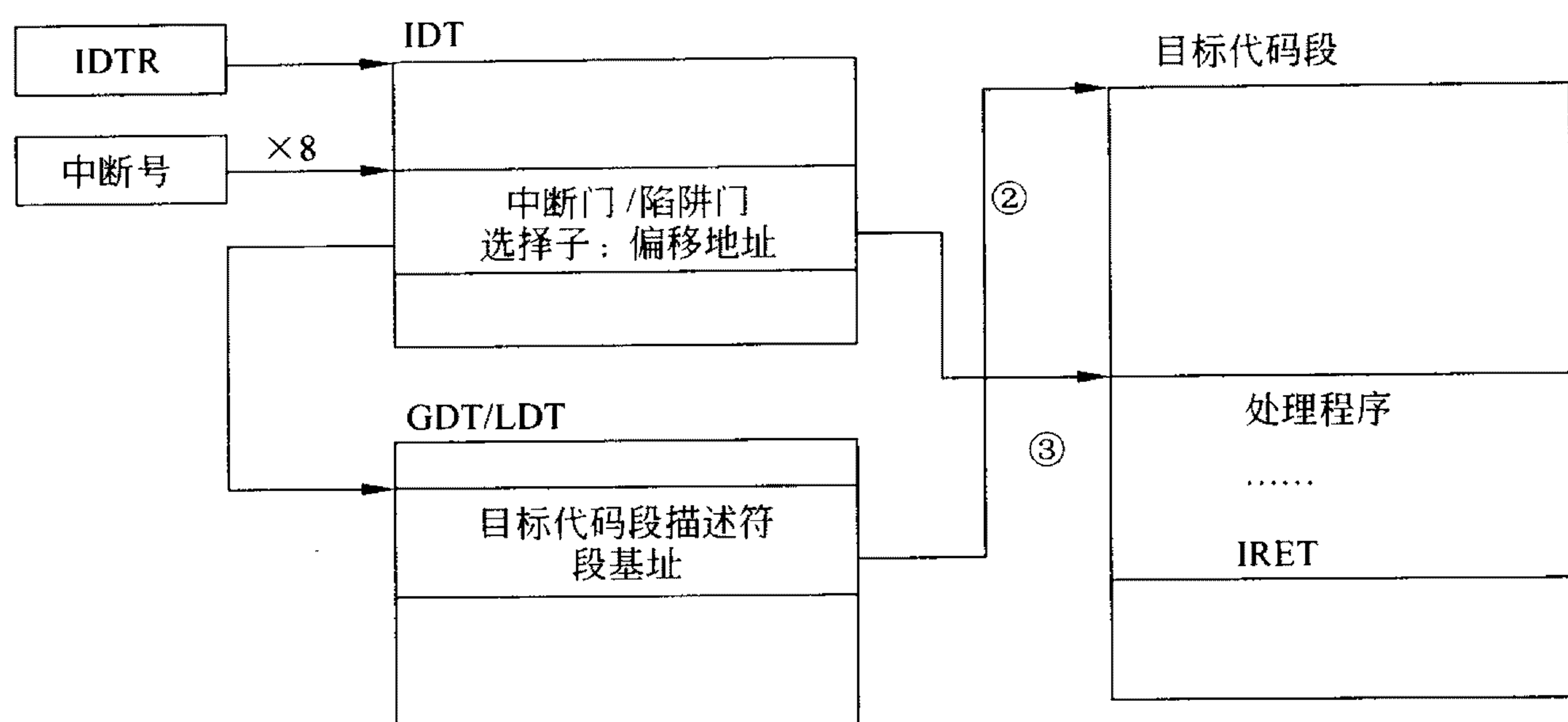


图 14-12 通过中断门或陷阱门的中断/异常处理过程

③ 将标志寄存器 EFLAGS 中的 NT 位置 0, 表示处理程序在利用中断返回指令 IRET 返回时, 返回到同一任务而不是一个嵌套任务。

④ 通过中断门的转移和通过陷阱门的转移之间的差别只是对 IF 标志的处理。对于中断门, 在转移过程中把 IF 置为 0, 使得在处理程序执行期间屏蔽掉 INTR 中断(当然, 在中断处理程序中可以为人为设置 IF 标志打开中断, 以使得在处理程序执行期间允许响应可屏蔽中断); 对于陷阱门, 在转移过程中保持 IF 位不变, 即如果 IF 位原来是 1, 那么通过陷阱门转移到处理程序之后仍允许 INTR 中断。因此, 中断门最适宜于处理中断, 而陷阱门适宜于处理异常。

3. 通过任务门的中断/异常处理过程

如果中断号指示的门描述符是任务门描述符, 那么控制转移到一个作为独立的任务方式而出现的处理程序。任务门中的选择子是指向描述对应处理程序任务的 TSS 段的选择子, 即该选择子指向一个可用的 286TSS 或 386TSS。通过任务门的中断/异常处理过程与通过任务门的任务间转移很相似, 可参见 14.2.3 节。主要的区别是, 对于提供出错码的异常处理, 在完成任务切换之后, 把出错码压入新任务的堆栈中。

通过任务门的转移, 在进入中断或异常处理程序时, 标志寄存器 EFLAGS 中的 NT 位被置为 1, 表示是嵌套任务, 则 IRET 指令返回时, 沿 TSS 中的链接字段返回到最后一个被挂起的任务。因为通过任务门的中断/异常处理是任务切换, 因此可以从任何特权级切换到目标任务的任何特权级。

14.3.4 中断和异常处理后的返回

中断返回指令 IRET 用于从中断或异常处理程序的返回。该指令的执行根据标志寄存器 EFLAGS 中的 NT 位是否为 1 分为两种情形。

① NT 位为 0, 表示是当前任务内的返回。这种情况出现在通过中断门或陷阱门转入处理程序的返回时。CPU 从堆栈顶弹出返回指针 EIP 及 CS, 然后弹出 EFLAGS 值。

弹出的 CS 选择子的 RPL 字段,确定返回后的特权级。如需要特权级改变,则从高特权级堆栈中弹出低特权级堆栈的指针 ESP 及 SS 的值。

② NT 位为 1,表示是嵌套任务的返回。因为通过任务门转入处理程序时,NT 位已被置 1。当前 TSS 中的链接字段保存有前一任务的 TSS 的选择子,取出该选择子进行任务切换就完成了返回。

中断返回指令 IRET 不仅能够用于由中断/异常引起的嵌套任务的返回,而且也适用于由段间调用指令 CALL 通过任务门引起的嵌套任务的返回,因为在执行通过任务门进行任务切换的段间调用指令 CALL 时,标志寄存器中的 NT 位被置为 1,表示任务嵌套。而 RET 指令是不能实现任务间的切换返回的。

14.4 保护模式下的输入/输出保护

在实地址模式下,用户程序可以使用 I/O 指令直接对端口进行读写。但保护模式支持多任务,因此为了避免输入/输出冲突,CPU 采用 I/O 特权级 IOPL 和 I/O 许可位图的方法来对输入/输出进行保护。

1. I/O 特权级

CPU 标志寄存器中,有两位是输入输出特权级 IOPL(I/O Privilege Level)(参见本书 3.3 节)。只有当 CPL 的级别高于或等于 IOPL 时,CPU 才可以执行所有与 I/O 相关的指令和访问 I/O 端口。与 I/O 相关的指令除了 I/O 指令,还包括 CLI 关中断和 STI 关中断指令。由于这些指令与 I/O 有关,并且只有在满足所列条件时才可以执行,所以把它们称为 I/O 敏感指令。注意,这些 I/O 敏感指令在实地址模式下总是可执行的。

当 CPL 的级别低于 IOPL,执行 CLI 和 STI 指令将引起通用保护异常,而 I/O 指令是否能够被执行要根据访问的 I/O 地址及 I/O 许可位图的情况而定,如果条件不满足却执行,那么将引起出错码为 0 的通用保护异常。

由于每个任务使用各自的 EFLAGS 值和拥有自己的 TSS,所以每个任务可以有不同的 IOPL,并且可以定义不同的 I/O 许可位图。

2. I/O 许可位图

由于 IOPL 对所有的 I/O 端口地址的访问作了限制,这使得在特权级 3 执行的应用程序要么可以访问所有的 I/O 端口,要么不可以访问所有的 I/O 端口,显然有时不能满足实际需要。因此,在 IOPL 的基础上又采用了 I/O 许可位图。I/O 许可位图规定了 I/O 空间中的哪些地址可以被在任何特权级执行的程序所访问。

I/O 许可位图在任务状态段 TSS 中,由二进制位串组成。位串中的每一位依次对应一个 I/O 地址,位串的第 0 位对应 I/O 地址 0,位串的第 n 位对应 I/O 地址 n。如果位串中的第 m 位为 0,那么对应的 I/O 地址 m 可以被在任何特权级执行的程序访问;如果位串中的第 m 位为 1,那么对应的 I/O 地址 m 只能被特权级别和 IOPL 相等或更高的程序访问,否则将引起通用保护异常。需要注意的是 CLI 和 STI 指令的执行和 I/O 许可位图

无关。

一条 I/O 指令最多可对四个 I/O 端口地址进行访问。因此当一条 I/O 指令涉及多个 I/O 地址时,需要根据 I/O 位图决定是否可访问该 I/O 地址,只有这多个 I/O 地址所对应的 I/O 许可位图中的位都为 0 时,该 I/O 指令才能被正常执行,如果对应位中任一位为 1,就会引起通用保护异常。

当前任务使用的 I/O 许可位图存储在当前任务 TSS 中低端的 64KB 内。TSS 内偏移 66H 的字确定 I/O 许可位图的开始偏移。I/O 许可位图总以字节为单位存储,所以位串所含的位数总被认为是 8 的倍数。由于 80386 支持的 I/O 地址空间大小是 64KB,所以构成 I/O 许可位图的二进制位串最大长度是 64KB,即位图的有效部分最大为 8KB。一个任务实际需要使用的 I/O 许可位图大小通常要远小于这个数目。

3. I/O 许可访问的过程

保护模式下处理器在执行 I/O 指令时进行许可访问的具体过程如下:

- ① 若 CPL 的级别高于 IOPL,则直接转步骤⑧;
- ② 取得 I/O 位图开始偏移;
- ③ 计算 I/O 地址对应位所在字节在 I/O 许可位图内的偏移;
- ④ 计算位偏移以形成屏蔽码值,即计算 I/O 地址对应位在字节中的第几位;
- ⑤ 字节偏移加上位图开始偏移,再加 1,所得值与 TSS 界限比较,若越界,则产生出错码为 0 的通用保护故障;
- ⑥ 若不越界,则从位图中读出对应字节及下一个字节;
- ⑦ 把读出的两个字节与屏蔽码进行与运算,若结果不为 0 表示检查未通过,则产生出错码为 0 的通用保护故障;
- ⑧ 进行 I/O 访问。

由上述 I/O 许可检查的过程可见,当进行许可位检查时,CPU 总是从 I/O 许可位图中读取两个字节。目的是尽快地执行 I/O 许可检查。一方面,常常要读取 I/O 许可位图的两个字节,因为即使只要检查两个位,也可能需要读取两个字节。另一方面,最多检查四个连续的位,即最多也只需读取两个字节。这也是在判别是否越界时再加 1 的原因。为了避免在读取 I/O 许可位图的最高字节时产生越界,必须在 I/O 许可位图的最后填加一个全 1 的字节,即 0FFH。此全 1 的字节应填加在最后一个位图字节之后,TSS 界限范围之前,即让填加的全 1 字节在 TSS 界限之内。

14.5 操作系统类指令

本节介绍的这些指令,通常只在操作系统代码中使用,所以称为操作系统类指令。这些指令可分为三种:实地址模式和任何特权级下可执行的指令、实地址模式及特权级 0 下可执行的指令和只能在保护模式下执行的指令。

14.5.1 实地址模式和任何特权级下可执行的指令

1. 存储全局描述符表寄存器指令

格式:

SGDT 目标操作数

说明: 目标操作数是 48 位(6 字节)的存储器操作数。该指令的功能是把全局描述符表寄存器 GDTR 的内容存储到目标操作数。GDTR 中的 16 位界限存入目标操作数的低字, GDTR 中的 32 位基地址存入目标操作数的高双字。该指令对标志位没有影响。

2. 存储中断描述符表寄存器指令

格式:

SIDT 目标操作数

说明: 目标操作数是 48 位(6 字节)的存储器操作数。该指令的功能是把中断描述符表寄存器 IDTR 的内容存储到目标操作数。IDTR 中的 16 位界限存入目标操作数的低字, IDTR 中的 32 位基地址存入目标操作数的高双字。该指令对标志位没有影响。

14.5.2 实地址模式和在特权级 0 下可执行的指令

下列指令涉及设置关键的寄存器, 所以只能在实地址模式和保护模式的特权级 0 下执行。为了从初始时的实地址模式转入保护模式, 必须做基本的准备工作, 例如, 设置全局描述符表寄存器 GDTR 等, 因此这些指令也允许在实地址模式下执行。

在保护模式下, 如果当前特权级 CPL 不为 0, 执行这些指令将引起错误码为 0 的通用保护故障。在虚拟 8086 方式下, 因为其 CPL 为 3, 所以执行这些指令也将会引起出错码为 0 的通用保护故障。

1. 装载全局描述符表寄存器指令

格式:

LGDT 源操作数

说明: 源操作数是 48 位(6 字节)的内存操作数。该指令的功能是把源操作数装入到全局描述符表寄存器 GDTR 中。操作数的低字是以字节为单位的段界限, 高双字是段基地址。该指令对标志位没有影响。

2. 装载中断描述符表寄存器指令

格式:

LIDT 源操作数

说明：源操作数 48 位(6 字节)的内存操作数。该指令的功能是把存储器中的源操作数装载到中断描述符表寄存器 IDTR 中。操作数的低字是以字节为单位的段界限,高双字是段基地址。该指令对标志位没有影响。

3. 控制寄存器数据传送指令

格式：

MOV 目标操作数,源操作数

说明：控制寄存器数据传送指令实现 CPU 的控制寄存器和 32 位通用寄存器之间的数据传送。所以,目标操作数和源操作数可以是 80386 使用的三个控制寄存器和任一 32 位通用寄存器,但不能同时是控制寄存器。该指令对标志位没有影响。

4. 清任务切换标志指令

格式：

CLTS

说明：每当任务切换时,控制寄存器 CR0 的任务切换标志 TS 被自动置 1。CLTS 指令的功能是将 TS 标志清 0。

14.5.3 只能在保护模式下执行的指令

下面的指令只能在保护模式下执行,如果在实地址模式下执行这些指令,将引起非法操作码故障(中断号为 6)。

1. 装载局部描述符表寄存器指令

格式：

LLDT 源操作数

说明：源操作数可以是 16 位通用寄存器或内存操作数。该指令的功能是把源操作数中的内容作为指示局部描述符表 LDT 的选择子装入到 LDTR 寄存器。该指令不影响标志位。

源操作数给定的选择子应该指向 GDT 中的类型为 LDT 的描述符。若 CPL 不为 0,那么执行该指令将产生出错码为 0 的通用保护故障。若被装载的选择子不指向 GDT 中的描述符,或者描述符类型不是 LDT 描述符,那么产生通用保护故障,错误码由该选择子构成。

2. 存储局部描述符表寄存器指令

格式：

SLDT 目的操作数

说明：目的操作数可以是 16 位通用寄存器或内存操作数。该指令的功能是把局部描述符表寄存器 LDTR 的内容存储到目的操作数，即把指向当前任务 LDT 的选择子存储到目的操作数中。该指令不影响标志位。

3. 装载任务寄存器指令

格式：

LTR 源操作数

说明：源操作数可以是 16 位通用寄存器或内存操作数。该指令的功能是将源操作数作为指向 TSS 描述符的选择子装载到任务寄存器 TR。源操作数表示的选择子不能为空，必须索引位于 GDT 中的描述符，并且描述符类型必须是可用 TSS，该加载的 TSS 被处理器自动标为“忙”。该指令对标志没有影响。若 CPL 不为 0，那么执行该指令将产生错误码为 0 的通用保护故障。若被加载的选择子不指向 GDT 中的可用 TSS 描述符，那么产生通用保护故障，错误码由该选择子构成。

4. 存储任务寄存器指令

格式：

STR 目的操作数

说明：目的操作数可以是 16 位通用寄存器或内存操作数。该指令的功能是把 TR 所含的指向当前任务 TSS 描述符的选择子存储到目的操作数。该指令不影响标志位。

14.6 保护模式下的程序设计

本节给出的保护模式程序实例，可在 80386 及以上处理器和 DOS 操作系统的软硬件环境下运行。可用 TASM 汇编，用 TLINK 链接（有 32 位寻址的代码段程序需要带 32 位链接选项“/3”）。在运行这些实例时，不要安装使用扩展内存的驱动程序，以避免发生冲突。

14.6.1 实地址模式与保护模式切换

加电、复位之后，CPU 自动工作在实地址模式下，因此保护模式下的程序设计的一个重要的工作就是从实地址模式进入保护模式。本节通过一个实例，来介绍如何实现实地址模式与保护模式的切换，也说明了保护模式编程的基本方法。

【例 14.6.1】 将数据缓冲区 BUF 单元开始存放的字符串，传送到地址为 110000H 开始的内存单元，并在屏幕上显示该字符串。

【设计思路】

① 因为实地址模式下不能访问 100000H 以上的内存区域，只有在保护模式下才能访问到该指定区域，因此程序必须实现实地址模式与保护模式的切换。



② 为了能反映出 32 位代码段和 16 位代码段的切换,本例要求在 32 位代码段中实现数据的传送;在 16 位代码段中完成数据的屏幕显示。

③ 由于 DOS 是一个实地址模式下运行的操作系统,当程序切换到保护模式后,DOS 功能调用 INT 21H 不再有效。因此在保护模式下的屏幕显示用的是直接写屏的方法(请参见本书 5.4 节)。

④ 为了节省篇幅,把有关结构类型的定义、宏指令的定义和符号常量的定义等语句集合在一个文件 SCD. INC 中,以便为本章每个例题程序所使用。

程序的具体实现步骤是:

① 作切换到保护方式的准备;

② 切换到保护方式的一个 32 位代码段,把位于常规内存缓冲区中的数据传送到指定高端内存缓冲区;

③ 再变换到保护方式下的一个 16 位代码段,将缓冲区中的字符串直接填入显示缓冲区显示;

④ 切换回实地址模式。

【被包含文件 SCD. INC】

;名称:SCD. INC

;功能:符号常量等的定义

. 586P

;

;打开 A20 地址线

;

```
ENABLEA20    MACRO
                PUSH    AX
                IN      AL,92H
                OR      AL,00000010B
                OUT     92H,AL
                POP     AX
            ENDM
```

;

;关闭 A20 地址线

;

```
DISABLEA20    MACRO
                PUSH    AX
                IN      AL,92H
                AND     AL,11111101B
                OUT     92H,AL
                POP     AX
            ENDM
```

;

;16 位偏移的段间直接转移指令的宏定义(在 16 位代码段中使用)

```

;-----
JUMP16      MACRO  SELECTOR,OFFSET
              DB      0EAH      ;操作码
              DW      OFFSET    ;16 位偏移量
              DW      SELECTOR   ;段值或段选择子
              ENDM

```

;32 位偏移的段间直接转移指令的宏定义(在 32 位代码段中使用)

```

;-----
JUMP32      MACRO  SELECTOR,OFFSET
              DB      0EAH      ;操作码
              DD      OFFSET
              DW      SELECTOR   ;段值或段选择子
              ENDM

```

```

;-----
JUMP32      MACRO  SELECTOR,OFFSET
              DB      0EAH      ;操作码
              DW      OFFSET
              DW      0
              DW      SELECTOR   ;段值或段选择子
              ENDM

```

;16 位偏移的段间调用指令的宏定义(在 16 位代码段中使用)

```

;-----
CALL16      MACRO  SELECTOR,OFFSET
              DB      9AH      ;操作码
              DW      OFFSET    ;16 位偏移量
              DW      SELECTOR   ;段值或段选择子
              ENDM

```

;32 位偏移的段间调用指令的宏定义(在 32 位代码段中使用)

```

;-----
CALL32      MACRO  SELECTOR,OFFSET
              DB      9AH      ;操作码
              DD      OFFSET
              DW      SELECTOR   ;段值或段选择子
              ENDM

```

```

;-----
CALL32      MACRO  SELECTOR,OFFSET
              DB      9AH      ;操作码
              DW      OFFSET
              DW      0

```



```

                                DW      SELECTOR    ;段值或段选择子
                                ENDM

;-----
;存储段和系统段描述符结构类型定义
;-----
DESC          STRUC
LIMITL        DW      0          ;段界限(BIT0—15)
BASEL          DW      0          ;段基地址(BIT0—15)
BASEM          DB      0          ;段基地址(BIT16—23)
ATTRIBUTES    DB      0          ;段属性
LIMITH        DB      0          ;段界限(BIT16—19)(含段属性的高4位)
BASEH          DB      0          ;段基地址(BIT24—31)
DESC          ENDS

;-----
;门描述符结构类型定义
;-----
GATE          STRUC
OFFSETL        DW      0          ;32位偏移的低16位
SELECTOR       DW      0          ;选择子
DCOUNT         DB      0          ;双字计数
GTYPE          DB      0          ;类型
OFFSETH        DW      0          ;32位偏移的高16位
GATE          ENDS

;-----
;伪描述符结构类型定义(用于装入全局或中断描述符表寄存器)
;-----
PDESC         STRUC
LIMIT          DW      0          ;16位界限
BASE           DD      0          ;32位基地址
PDESC         ENDS

;-----
;任务状态段结构类型定义
;-----
TSS           STRUC
TRLINK         DW      0          ;链接字段
              DW      0          ;不使用,置为0
TRESP0         DD      0          ;0级堆栈指针
TRSS0          DW      0          ;0级堆栈段寄存器
              DW      0          ;不使用,置为0
TRESP1         DD      0          ;1级堆栈指针
TRSS1          DW      0          ;1级堆栈段寄存器
              DW      0          ;不使用,置为0
TRESP2         DD      0          ;2级堆栈指针

```

TRSS2	DW	0	;2 级堆栈段寄存器
	DW	0	;不使用,置为 0
TRCR3	DD	0	;CR3
TREIP	DD	0	;EIP
TREFLAG	DD	0	;EFLAGS
TREAX	DD	0	;EAX
TRECX	DD	0	;ECX
TREDX	DD	0	;EDX
TREBX	DD	0	;EBX
TRESP	DD	0	;ESP
TREBP	DD	0	;EBP
TRESI	DD	0	;ESI
TREDI	DD	0	;EDI
TRES	DW	0	;ES
	DW	0	;不使用,置为 0
TRCS	DW	0	;CS
	DW	0	;不使用,置为 0
TRSS	DW	0	;SS
	DW	0	;不使用,置为 0
TRDS	DW	0	;DS
	DW	0	;不使用,置为 0
TRFS	DW	0	;FS
	DW	0	;不使用,置为 0
TRGS	DW	0	;GS
	DW	0	;不使用,置为 0
TRLDTR	DW	0	;LDTR
	DW	0	;不使用,置为 0
TRTRIP	DW	0	;调试陷阱标志(只用位 0)
TRIOMAP	DW	\$ +2	;指向 I/O 许可位图区的段内偏移
TSS	ENDS		
;-----			
;存储段描述符类型值说明			
;-----			
ATDR	=	90H	;存在的只读数据段类型值
ATDW	=	92H	;存在的可读写数据段属性值
ATDWA	=	93H	;存在的已访问可读写数据段类型值
ATCE	=	98H	;存在的只执行代码段属性值
ATCER	=	9AH	;存在的可执行可读代码段属性值
ATCCO	=	9CH	;存在的只执行一致代码段属性值
ATCCOR	=	9EH	;存在的可执行可读一致代码段属性值
;-----			
;系统段描述符类型值说明			
;-----			

ATLDT	=	82H	;局部描述符表段类型值
ATTASKGATE	=	85H	;任务门类型值
AT386TSS	=	89H	;可用 386 任务状态段类型值
AT386CGATE	=	8CH	;386 调用门类型值
AT386IGATE	=	8EH	;386 中断门类型值
AT386TGATE	=	8FH	;386 陷阱门类型值

;

;DPL 值说明

;

DPL0	=	00H	;DPL=0
DPL1	=	20H	;DPL=1
DPL2	=	40H	;DPL=2
DPL3	=	60H	;DPL=3

;

;RPL 值说明

;

RPL0	=	00H	;RPL=0
RPL1	=	01H	;RPL=1
RPL2	=	02H	;RPL=2
RPL3	=	03H	;RPL=3

;

;IOPL 值说明

;

IOPL0	=	0000H	;IOPL=0
IOPL1	=	1000H	;IOPL=1
IOPL2	=	2000H	;IOPL=2
IOPL3	=	3000H	;IOPL=3

;

;其他常量值说明

;

D32	=	40H	;32 位代码段标志
GL	=	80H	;段界限以 4K 为单位标志
TIL	=	04H	;TI=1(局部描述符表标志)

;

【程序清单】

;功能:演示实方式和保护方式切换

INCLUDE SCD. INC

;

DSEG SEGMENT USE16 ;16 位数据段

;

GDT	LABEL	BYTE	;全局描述符表
DUMMY	DESC	<>	;空描述符
NORMAL	DESC	<0FFFFH,,,ATDW,,>	;规范段描述符

CODE32	DESC	<0FFFFH,,,ATCE,D32,>	;32 位代码段描述符
CODE16	DESC	<0FFFFH,,,ATCE,,>	;16 位代码段描述符
DATAS	DESC	<BUFLen-1,,,ATDR,,>	;源数据段描述符
DATAD	DESC	<BUFLen-1,0,11H,ATDW,,>	;目标数据段描述符
DATAx	DESC	<3999,8000H,0BH,ATDW,,>	;显示存储区描述符
;-----			
GDTLEN	=	\$-GDT	;全局描述符表长度
VGDTR	PDESC	<GDTLEN-1,>	;伪描述符
;-----			
NORMAL_SEL	=	NORMAL-GDT	;规范段描述符选择子
CODE32_SEL	=	CODE32-GDT	;32 位代码段选择子
CODE16_SEL	=	CODE16-GDT	;16 位代码段选择子
DATAS_SEL	=	DATAS-GDT	;源数据段选择子
DATAD_SEL	=	DATAD-GDT	;目标数据段选择子
DATAx_SEL	=	DATAx-GDT	;显示存储区数据段选择子
;-----			
BUFFER	DB	'WELCOME TO PROTECT WORLD !'	;源缓冲区
BUFLen	EQU	\$-BUFFER	;源缓冲区字节长度
;-----			
DSEG	ENDS		;数据段定义结束
;-----			
CSEG1	SEGMENT USE16	'REAL'	;16 位代码段
	ASSUME	CS:CSEG1,DS:DSEG	
START	PROC		
	MOV	AX,DSEG	
	MOV	DS,AX	
		;准备要加载到 GDTR 的伪描述符	
	MOV	BX,16	
	MUL	BX	
	ADD	AX,OFFSET GDT	;计算并设置基地址
	ADC	DX,0	;界限已在定义时设置好
	MOV	WORD PTR VGDTR.BASE,AX	
	MOV	WORD PTR VGDTR.BASE+2,DX	
		;设置 32 位代码段描述符	
	MOV	AX,CSEG2	
	MUL	BX	
	MOV	WORD PTR CODE32.BASEL,AX	
	MOV	BYTE PTR CODE32.BASEM,DL	
	MOV	BYTE PTR CODE32.BASEH,DH	
		;设置 16 位代码段描述符	
	MOV	AX,CSEG3	
	MUL	BX	
	MOV	WORD PTR CODE16.BASEL,AX	;代码段开始偏移为 0
	MOV	BYTE PTR CODE16.BASEM,DL	;代码段界限已在定义时设置好



```

MOV      BYTE PTR CODE16. BASEH, DH
;设置源数据段描述符
MOV      AX, DS
MUL      BX                      ;计算并设置源数据段基址
ADD      AX, OFFSET BUFFER
ADC      DX, 0
MOV      WORD PTR DATAS. BASEL, AX
MOV      BYTE PTR DATAS. BASEM, DL
MOV      BYTE PTR DATAS. BASEH, DH
;加载 GDTR
LGDT     VGDTR
CLI                      ;关中断
ENABLEA20                ;打开地址线 A20
;切换到保护方式
MOV      EAX, CR0
OR       AL, 1
MOV      CR0, EAX
;清指令预取队列,并真正进入保护方式
JUMP16 CODE32_SEL, <HIGH OFFSET SPM32>
TOREAL: ;现在又回到实方式
MOV      AX, DSEG
MOV      DS, AX
DISABLEA20                ;关闭地址线 A20
STI
MOV      AH, 4CH
INT      21H
START    ENDP
CSEG1    ENDS                      ;代码段定义结束
;-----
CSEG2    SEGMENT USE32 'PM32'
ASSUME   CS: CSEG2
SPM32    PROC
MOV      AX, DATAS_SEL      ;装载源数据段选择子
MOV      DS, AX
MOV      AX, DATAD_SEL      ;装载目的数据段选择子
MOV      ES, AX
XOR      ESI, ESI
XOR      EDI, EDI
MOV      ECX, BUFLen
CLD
REP      MOVSB              ;数据传送
;变换到 16 位代码段
JUMP32   CODE16_SEL, <OFFSET SPM16>
SPM32    ENDP

```

```

C32LEN      =          $
CSEG2       ENDS

;-----
CSEG3       SEGMENT USE16 'PM16'
            ASSUME  CS,CSEG3

SPM16       PROC
            MOV     AX,DATAD_SEL      ;设置指针和计数器
            MOV     DS,AX
            MOV     AX,DATAX_SEL
            MOV     ES,AX
            XOR     SI,SI
            XOR     DI,DI
            MOV     AH,7              ;显示属性为黑底白字
            MOV     CX,BUFLEN
AGAIN:      LODSB
            STOSW                    ;将显示字符直接写屏
            LOOP    AGAIN
            MOV     AX,NORMAL_SEL     ;将 NORMAL 段选择子装入 DS 和 ES
            MOV     DS,AX
            MOV     ES,AX
            MOV     EAX,CR0
            AND     AL,11111110B
            MOV     CR0,EAX
            JMP     FAR PTR TOREAL    ;切换到实地址模式

SPM16       ENDP
CSEG3       ENDS
            END     START

```

【程序分析】

① 程序一开始使用带 p 的方式选择伪指令 .586p, 表示程序中可以用特权级指令。如 MOV CR0,EAX。

② 切换到保护方式的准备工作。

在从实地址模式切换到保护模式之前, 必须作一些准备工作。最基本的准备工作是建立合适的全局描述符表, 并使用 GDTR 指向该 GDT。因为在切换到保护方式时, 至少要把代码段的选择子装载到 CS, 所以应 GDT 中至少含有代码段的描述符。

从本实例源程序可见, 全局描述符表 GDT 有七个描述符。第一个是空描述符; 第二个是规范段描述符; 第三个和第四个分别为 32 位代码段和 16 位代码段描述符; 第五个、第六个和第七个分别是源数据段、目标数据段和显示存储区段描述符。本实例各描述符中的段界限是在定义时设置的, 并且除伪描述符 VGDTR 中的界限按 GDT 的实际长度设置外, 各使用的存储段描述符的界限大都规定为 0FFFFH。另外, 描述符中的段属性也根据所描述段的类型被预置。

由于在切换到保护方式后就要引用 GDT, 所以在切换到保护方式前必须装载

GDTR。实例中使用如下指令装载 GDTR:

```
LGDT VGDTR
```

该指令的功能是把存储器中的伪描述符 VGDTR 装入到全局描述符表寄存器 GDTR 中。伪描述符 VGDTR 的结构如前所述结构类型 PDESC 所示,低字是以字节为单位的全局描述符表段的界限,高双字为描述符表段的线性基地址(本实例不启用分页机制,所以线性地址等同于物理地址)。

③ 由实地址模式切换到保护模式。在做好准备后,从实地址模式切换到保护模式。原则上只要把控制寄存器 CR0 中的 PE 位置 1 即可。本实例采用如下三条指令设置 PE 位:

```
MOV EAX,CR0
OR EAX,1
MOV CR0,EAX
```

执行上面的三条指令后,处理器转入保护模式,但 CS 中的内容还是实地址模式下代码段的段值,而不是保护模式下代码段的选择子,所以在取指令之前得把代码段的选择子装入 CS。为此,紧接着这三条指令,安排一条如下所示的段间转移指令:

```
JUMP16 Code32_Sel,<high OFFSET SPM32>
```

这条段间转移指令在实地址模式下被预取并在保护方式下被执行。该指令必须用 high 运算符取得 32 位代码段的 16 位入口偏移。利用这条段间转移指令可把保护模式下代码段的选择子装入 CS,同时也刷新指令预取队列。从此真正进入保护模式。

④ 打开和关闭地址线 A20。PC 及其兼容机的第 21 根地址线(A20)较特殊,计算机系统中一般安排一个“门”控制该地址线是否有效。为了访问地址在 1MB 以上的存储单元,应先打开控制地址线 A20 的“门”。这种设置与实地址模式下只使用最低端的 1MB 存储空间有关,与处理器是否工作在实地址模式或保护方式无关,即使在关闭地址线 A20 时,也可进入保护模式。如何打开和关闭地址线 A20 与计算机系统的具体设置有关。在本例中定义了两个宏,打开地址线 A20 的宏 EnableA20 和关闭地址线 A20 的宏 DisableA20,这两个宏指令在一般的 PC 兼容机上都是可行的。

⑤ 由保护模式切换到实地址模式。在 80386 上,从保护模式切换到实地址模式的过程类似于从实地址模式切换到保护模式。原则上只要把控制寄存器 CR0 中的 PE 位清零即可。实际上,在此之后也要安排一条段间转移指令,一方面清零指令预取队列,另一方面把实地址模式下代码段的段值送 CS。这条段间转移指令在保护方式下被预取并在实地址模式下被执行。

⑥ 32 位代码段与 16 位代码段之间的切换。在保护模式下,通过如下直接段间转移指令从 32 位代码段切换到 16 位代码段:

```
JUMP32 Code16_Sel,<OFFSET SPM16>
```

从该宏指令的定义可知,该转移指令含 48 位指针,其高 16 位是 16 位代码段的选择

子,低 32 位是 16 位代码段的入口偏移。该指令在 32 位方式下预取并执行。由于在 32 位方式下执行,所以要使用 48 位指针。

⑦ 保护模式下的数据传送。在 32 位代码段中,默认的操作数大小是 32 位,默认的存储单元地址大小是 32 位。由于串操作指令使用的指针寄存器是 ESI 和 EDI,LOOP 指令使用的计数器是 ECX,所以,在代码段 CSEG2 中,为了使用串操作指令,对 ESI 和 EDI 等寄存器应赋初值。

⑧ 从本实例的 GDT 中可见,两个数据段的界限都是根据实际大小而设置的。从源程序代码段 CSEG3 可见,在切换到实地址模式之前,必须把一个指向规范段的描述符 Normal 的选择子装载到 DS 和 ES。这是因为在 14.1.1 节分段管理中已介绍过,每个段寄存器都配有段描述符高速缓冲寄存器。在实地址模式下,每个段也对应一个段高速缓冲寄存器,每个段的 32 位段界限都固定为 0FFFFH,段属性的许多位也是固定的,工作在特权级 0。因此,在准备结束保护模式回到实地址模式之前,要加载一个合适的描述符选择子到有关段寄存器,使得对应段描述符高速缓冲寄存器中含有合适的段界限和属性。本实例 GDT 中的描述符 Normal 就是这样一个描述符,在返回实地址模式之前把对应选择子 Normal_Sel 加载到 DS 和 ES 就是这个目的。16 位代码段描述符中的内容已符合实地址模式的需要。需要注意的是,不能从 32 位代码段返回实地址模式,这是因为无法实现从 32 位代码段返回时,CS 高速缓冲寄存器中的属性符合实地址模式的要求(实地址模式不能改变段属性)。

⑨ 作为第一个实地址模式和保护模式切换的例子,本实例作了大量的简化处理。通常,由实地址模式切换到保护模式的准备工作还应包含建立中断描述符表。但本实例没有建立中断描述符表。为此,要求整个过程在关中断的情况下进行;要求不使用软中断指令;假设不发生任何异常。否则会导致系统崩溃。

本实例未使用局部描述符表,所以在进入保护模式后没有设置局部描述符表寄存器 LDTR。为此,在保护模式下使用的段选择子都指定 GDT 中的描述符。

本实例各描述符特权级 DPL 和各选择子的请求特权级 RPL 均为 0,在保护方式下运行时的当前特权级 CPL 也是 0。

14.6.2 保护模式下中断和异常程序设计

在保护模式下,CPU 把外部中断(硬件中断)称为“中断”,而把内部中断称为“异常”。CPU 根据中断号从中断描述表 IDT 中取得相应的门描述符,从而获得中断或异常处理程序的入口地址。本节通过一个实例,来介绍保护模式下中断和异常处理程序设计。

【例 14.6.2】 利用日时钟中断实现在屏幕的左上角每秒显示一行字符串“WELCOME TO PROTECT WORLD!”,显示 10 次后程序结束。

【设计思路】 日时钟中断的工作原理请参考本书 9.7 节。为了在保护模式下响应中断和处理异常,程序中必须有中断描述符表 IDT。IDT 含有 256 个门描述符。8 号安排的是一个通向日时钟中断处理程序的中断门。为了说明陷阱异常的处理,0FEH 号安排的是通向显示处理程序的陷阱门,其他均安排成通向其他中断或异常处理程序的陷阱门。GDT 中除了含有常见的几个描述符外,还含有描述时钟中断处理程序所使用的代

码段和数据段描述符,以及描述显示程序所使用的代码段和数据段描述符。

【程序清单】

;功能:演示中断处理的实现

INCLUDE SCD. INC

GDTSEG SEGMENT PARA USE16 ;全局描述符表数据段(16 位)

 ;全局描述符表 GDT
GDT LABEL BYTE
DUMMY DESC <> ;空描述符
NORMAL DESC <0FFFFH,,,ATDW,,> ;规范段描述符
VIDEOBUF DESC <0FFFFH,8000H,0BH,ATDW,,> ;显示存储区段描述符

EFFGDT LABEL BYTE
 ;演示代码段描述符
DEMOCODE DESC <0FFFFH,DEMOCODESEG,,ATCE,,>
 ;演示数据段描述符
DEMODATA DESC <DEMODATALEN-1,DEMODATASEG,,ATDW,,>
 ;演示堆栈段描述符
DEMOSTACK DESC <DEMOSTACKLEN-1,DEMOSTACKSEG,,ATDWA,,>
 ;0FEH 号中断处理程序(显示程序)代码段描述符
ECHOCODE DESC <ECHOCODELEN-1,ECHOCODESEG,,ATCE,,>
 ;0FEH 号中断处理程序(显示程序)数据段描述符
ECHODATA DESC <ECHODATALEN-1,ECHODATASEG,,ATDW,,>
 ;8 号中断处理程序代码段描述符
TICODE DESC <TICODELEN-1,TICODESEG,,ATCE,,>
 ;8 号中断处理程序数据段描述符
TIDATA DESC <TIDATALEN-1,TIDATASEG,,ATDW,,>
 ;其他中断或异常处理程序代码段描述符
OTHER DESC <OTHERCODELEN-1,OTHERCODESEG,,ATCE,,>

GDTLEN = \$ - GDT ;全局描述符表长度
GDNUM = (\$ - EFFGDT)/(SIZE DESC) ;需特殊处理的描述符数

NORMAL_SEL = NORMAL - GDT ;规范段描述符选择子
VIDEO_SEL = VIDEOBUF - GDT ;视频缓冲区段描述符选择子

DEMOCODE_SEL = DEMOCODE - GDT ;演示代码段的选择子
DEMODATA_SEL = DEMODATA - GDT ;演示数据段的选择子
DEMOSTACK_SEL = DEMOSTACK - GDT ;演示堆栈段的选择子
ECHOCODE_SEL = ECHOCODE - GDT ;0FEH 号中断程序代码段选择子
ECHODATA_SEL = ECHODATA - GDT ;0FEH 号中断程序数据段选择子
TICODE_SEL = TICODE - GDT ;8 号中断程序代码段选择子

```

TIDATA_SEL      =      TIDATA-GDT      ;8 号中断程序数据段选择子
OTHER_SEL       =      OTHER-GDT       ;其他中断或异常代码段选择子
;-----
GDTSEG          ENDS                    ;全局描述符表段定义结束
;-----
IDTSEG          SEGMENT PARA USE16      ;中断描述符表数据段(16 位)
;-----
IDT              LABEL  BYTE            ;中断描述符表
                ;0--7 的 8 个陷阱门描述符
                REPT    8
                GATE    <OTHERBEGIN,OTHER_SEL,,AT386TGATE,>
                ENDM
                ;对应 8 号(时钟)中断处理程序的门描述符
                GATE    <TIBEGIN,TICODE_SEL,,AT386IGATE,>
                ;从 9--0FDH 的 245 个陷阱门描述符
                REPT    245
                GATE    <OTHERBEGIN,OTHER_SEL,,AT386TGATE,>
                ENDM
                ;对应 0FEH 号中断处理程序的陷阱门描述符
                GATE    <ECHOBEGIN,ECHOCODE_SEL,,AT386TGATE,>
                ;对应 0FFH 号中断处理程序的陷阱门描述符
                GATE    <OTHERBEGIN,OTHER_SEL,,AT386TGATE,>
;-----
IDTLEN          =      $-IDT
;-----
IDTSEG          ENDS                    ;中断描述符表段定义结束
;-----
;其他中断或异常处理程序的代码段
;-----
OTHERCODESEG    SEGMENT PARA USE16
                ASSUME CS:OTHERCODESEG
;-----
OTHERBEGIN      PROC    FAR
                MOV     AX,VIDEO_SEL
                MOV     ES,AX
                MOV     AH,17H          ;在屏幕左上角显示蓝底白字
                MOV     AL,'!'          ;符号"!"
                MOV     WORD PTR ES:[0],AX
                JMP     $                ;无限循环
OTHERBEGIN      ENDP
;-----
OTHERCODELEN    =      $
OTHERCODESEG    ENDS
;-----

```

;8号中断处理程序的数据段

```

TIDATASEG      SEGMENT PARA USE16
COUNT        DB      18                ;中断计数初值
TIDATALEN      =        $
TIDATASEG      ENDS

```

;8号中断处理程序的代码段

```

TICODESEG      SEGMENT PARA USE16
                ASSUME CS:TICODESEG,DS:TIDATASEG
;-----
TIBEGIN        PROC      FAR
                PUSH      EAX                ;保护现场
                PUSH      DS
                PUSH      FS
                PUSH      GS
                MOV        AX,TIDATA_SEL      ;置中断处理程序数据段
                MOV        DS,AX
                MOV        AX,ECHODATA_SEL    ;置显示过程数据段
                MOV        FS,AX
                MOV        AX,DEMODATA_SEL    ;置演示程序数据段
                MOV        GS,AX
                CMP        COUNT,0
                JNZ         TI2                ;计数非0表示未到1秒
                MOV        COUNT,18          ;每秒约18次
                INT         0FEH              ;调用0FEH号中断处理程序显示
                CMP        BYTE PTR FS:MESS,10 ;是否已显示10行字符串?
                JNZ         TI1
                MOV        BYTE PTR GS:FLAG,1 ;已显示10行字符串时置标记
TI1:           INC         BYTE PTR FS:MESS    ;调整显示下一行
TI2:           DEC         COUNT              ;调整计数
                POP        GS                ;恢复现场
                POP        FS
                POP        DS
                MOV        AL,20H              ;通知中断控制器中断处理结束
                OUT         20H,AL
                POP        EAX
                IRETD        ;中断返回
TIBEGIN        ENDP
;-----
TICODELEN      =        $
TICODESEG      ENDS
;-----

```

;0FEH 号中断处理程序数据段

;

ECHODATASEG SEGMENT PARA USE16

MESS DB 0

MESG DB 'WELCOME TO PROTECT WORLD !'

MESGLEN EQU \$ - MESG

ECHODATALEN = \$

ECHODATASEG ENDS

;

;0FEH 号中断处理程序(显示程序)的代码段

;

ECHOCODESEG SEGMENT PARA USE16

ASSUME CS:ECHOCODESEG,DS:ECHODATASEG

;

ECHOBEGIN PROC FAR

PUSH AX ;保护现场

PUSH SI

PUSH DI

PUSH DS

PUSH ES

MOV AX,ECHODATA_SEL ;置显示过程数据段

MOV DS,AX

MOV AX,VIDEO_SEL ;置显示存储区数据段

MOV ES,AX

MOV SI,OFFSET MESG ;置指针和计数器

XOR DI,DI

MOV AL,80 * 2

MUL MESS

ADD DI,AX ;每次显示在屏幕的下一行

MOV AH,4EH

MOV CX,MESGLEN

AGAIN:

LODSB

STOSW ;显示字符串

LOOP AGAIN

POP ES

POP DS

POP DI

POP SI

POP AX

IRETD

ECHOBEGIN ENDP

;

ECHOCODELEN = \$

ECHOCODESEG ENDS

```

;-----
;演示任务的堆栈段
;-----
DEMOSTACKSEG SEGMENT PARA USE16
DEMOSTACKLEN =      1024
                DB      DEMOSTACKLEN DUP(0)
DEMOSTACKSEG ENDS
;-----
;演示任务的数据段
;-----
DEMODATASEG  SEGMENT PARA USE16
FLAG          DB      0
DEMODATALEN  =      $
DEMODATASEG  ENDS
;-----
;演示任务的代码段
;-----
DEMOCODESEG  SEGMENT  PARA USE16
                ASSUME  CS:DEMOCODESEG,DS:DEMODATASEG
;-----
DEMOBEGIN    PROC    FAR
                MOV     AX,DEMOSTACK_SEL    ;置堆栈
                MOV     SS,AX
                MOV     SP,DEMOSTACKLEN    ;置数据段
                MOV     AX,DEMODATA_SEL
                MOV     DS,AX
                MOV     ES,AX
                MOV     FS,AX
                MOV     GS,AX
                MOV     AL,11111110B        ;置中断屏蔽字
                OUT     21H,AL              ;打开日时钟中断
                STI                      ;开中断
DEMOCONTI:    CMP     BYTE PTR FLAG,0      ;判结束标志
                JZ      DEMOCONTI          ;直到不为0
                CLI                      ;关中断
                ;准备回实方式
TODOS:        MOV     AX,NORMAL_SEL        ;恢复实方式段描述符高速缓存
                MOV     DS,AX
                MOV     ES,AX
                MOV     FS,AX
                MOV     GS,AX
                MOV     SS,AX
                MOV     EAX,CR0            ;准备返回实地址模式
                AND     AL,11111110B

```



```

        MOV     CR0,EAX
        JUMP16  <SEG REAL>,<OFFSET REAL>
DEMOBEGIN    ENDP
;-----
DEMOCODELEN  =      $
DEMOCODESEG  ENDS
;=====
RDATASEG     SEGMENT PARA USE16           ;实方式数据段
VGDTTR       PDESC    <GDTLEN-1,>        ;GDT 伪描述符
VIDTR        PDESC    <IDTLEN-1,>        ;IDT 伪描述符
NORVIDTR     PDESC    <3FFH,>            ;用于保存原 IDTR 值
SPVAR        DW       ?                  ;用于保存实方式下的 SP
SSVAR        DW       ?                  ;用于保存实方式下的 SS
IMASKREGV    DB       ?                  ;用于保存原中断屏蔽寄存器值
RDATASEG     ENDS
;-----
RCODESEG     SEGMENT PARA USE16           ;实方式代码段
        ASSUME CS:RCODESEG,DS:RDATASEG
;-----
START        PROC
        MOV     AX,RDATASEG
        MOV     DS,AX
        CLD
        CALL    INITGDT                  ;初始化全局描述符表 GDT
        CALL    INITIDT                  ;初始化中断描述符表 IDT
        MOV     SSVAR,SS                  ;保存堆栈指针
        MOV     SPVAR,SP
        SIDT    NORVIDTR                  ;保存 IDTR
        IN      AL,21H
        MOV     BYTE PTR IMASKREGV,AL
        LGDT    VGDTTR                    ;装载 GDTR
        CLI                      ;关中断
        LIDT    VIDTR                      ;装载 IDTR
        MOV     EAX,CR0
        OR      AL,1
        MOV     CR0,EAX
        JUMP16  <DEMOCODE_SEL>,<OFFSET DEMOBEGIN>
                                           ;转演示任务
REAL:
        MOV     AX,RDATASEG
        MOV     DS,AX
        LSS     SP,DWORD PTR SPVAR ;又回到实方式
        LIDT    NORVIDTR
        MOV     AL,IMASKREGV

```

```

                                OUT    21H,AL
                                STI
                                MOV    AX,4C00H
                                INT     21H
START                           ENDP
;-----
INITGDT   PROC
                                PUSH   DS
                                MOV     AX,GDTSEG
                                MOV     DS,AX
                                MOV     CX,GDNUM
                                MOV     SI,OFFSET EFFGDT
INITG:    MOV     AX,[SI].BASEL
                                MOVZX   EAX,AX
                                SHL     EAX,4
                                SHLD    EDX,EAX,16
                                MOV     WORD PTR [SI].BASEL,AX
                                MOV     BYTE PTR [SI].BASEM,DL
                                MOV     BYTE PTR [SI].BASEH,DH
                                ADD     SI,SIZE DESC
                                LOOP    INITG
                                POP     DS
                                MOV     BX,16
                                MOV     AX,GDTSEG
                                MUL     BX
                                MOV     WORD PTR VGDTR.BASE,AX
                                MOV     WORD PTR VGDTR.BASE+2,DX
                                RET
INITGDT   ENDP
;-----
INITIDT   PROC
                                MOV     BX,16
                                MOV     AX,IDTSEG
                                MUL     BX
                                MOV     WORD PTR VIDTR.BASE,AX
                                MOV     WORD PTR VIDTR.BASE+2,DX
                                RET
INITIDT   ENDP
;-----
RCODESEG  ENDS
                                END     START

```

【程序分析】**(1) 实方式下初始化 GDT 和 IDT**

GDT 和 IDT 中的描述符的界限和属性值都在定义时预置。为了方便,定义时把各段的段值存放在相应描述符的段基址低 16 位字段中。由于实例中各段在实地址模式下定义,所以把段值乘 16 就是对应的段基址。

(2) 时钟中断仍使用 8H 号中断向量

实例使用了日时钟作为外部中断源,没有通过重新设置中断控制器的方法改变对应的中断向量。所以,时钟中断使用的 8H 号中断号就与双重故障异常使用的中断号发生冲突。但实例仅是演示程序,所以只要保证不发生双重故障异常,就可避免冲突,从而不会影响演示。程序中设置中断屏蔽寄存器,仅开放时钟中断。所以,在开中断状态下,只可能发生时钟中断,而不会发生其他外部中断。

(3) 时钟中断处理程序的设计

由于通过中断门转入时钟中断处理程序,所以在控制转移时不发生任务切换。但外部中断随时可能发生,因此中断处理程序必须采取保护现场等措施。作为演示程序,该中断处理程序对日时钟中断进行计数,满 18 次后,就认为已满一秒,这时调用 INT 0FEH 于显示字符串信息。

(4) 利用一个软中断(陷阱处理)程序实现显示

为了演示陷阱及其处理,把显示过程安排成陷阱处理程序。上述时钟中断处理程序通过软中断调用指令 INT 调用该显示程序。在控制转移时,也没有任务切换。该陷阱处理程序相当于一个“软中断”处理程序,类似于实地址模式下的 BIOS 中断 INT 10H。

(5) 对其他中断或异常的响应

为了简单,除了 8H 号和 0FEH 号外,IDT 中其他的门均通向一个处理程序。该处理程序用于处理其他中断或异常。处理过程也极其简单,在屏幕左上角显示蓝底白字的符号“!”,然后进入无限循环。

(6) 没有特权级变换

为了简单,实例涉及的中断处理程序和异常处理程序都保持特权级 0。所以,控制转移时不发生特权级变换。因此,没有使用其他堆栈。

(7) 装载和保存 IDTR 寄存器

使 IDT 发挥作用之前,还要装载中断描述符表寄存器 IDTR;但为了回到实地址模式后,恢复原来的 IDTR 之内容,应先保存 IDTR 的内容。

14.6.3 输入/输出保护及任务切换

保护模式支持多任务,采用 I/O 特权级 IOPL 和 I/O 许可位图的方法来对输入/输出进行保护。本节通过一个实例,来介绍保护模式下对端口进行读写的程序设计及任务的切换。

【例 14.6.3】 通过任务门调用输入/输出测试任务,在测试任务中使 PC 系列机的 8254 计数器产生固定频率的方波,使扬声器发声。

【设计思路】 8254 芯片的工作原理请参考本书 8.3 节。本例使用段间调用指令

CALL 通过任务门调用输入/输出测试任务,在测试任务中演示 I/O 许可位图对端口读写指令的作用。

具体步骤如下:

- ① 在实地址模式下做好必要准备后,切换到保护模式;
- ② 进入保护模式的临时代码段后,把演示任务的 TSS 段描述符装入 TR,并设置演示任务的堆栈;
- ③ 进入演示代码段,演示代码段的特权级是 0;
- ④ 通过任务门调用测试任务,使扬声器发声;
- ⑤ 从演示代码转临时代码,准备返回实地址模式;
- ⑥ 返回实地址模式,并作结束处理。

【程序清单】

;功能:演示 I/O 保护及任务切换

INCLUDE SCD.INC

```

;-----
GDTSEG      SEGMENT PARA USE16                      ;全局描述符表段(16 位)
GDT          LABEL      BYTE
              ;空描述符
DUMMY        DESC      <>
              ;规范段描述符及选择子
NORMAL       DESC      <0FFFFH,,,ATDW,,>
NORMAL_SEL =          NORMAL-GDT
              ;视频缓冲区段描述符(DPL=3)及选择子(任何特权级可写)
;-----
EFFGDT       LABEL      BYTE
;演示任务 TSS 段描述符及选择子
DEMOTSS      DESC      <DEMOTSSLEN-1,DEMOTSSSEG,,AT386TSS,,>
DEMOTSS_SEL =          DEMOTSS-GDT
              ;演示任务堆栈段描述符及选择子
DEMOSTACK    DESC      <DEMOSTACKLEN-1,DEMOSTACKSEG,,ATDW,D32,>
DEMOSTACK_SEL =        DEMOSTACK-GDT
              ;演示代码段描述符及选择子
DEMOCODE     DESC      <0FFFFH,DEMOCODESEG,,ATCE,D32,>
DEMOCODE_SEL =        DEMOCODE-GDT
              ;属于演示任务的临时代码段描述符及选择子
TEMPCODE     DESC      <0FFFFH,TEMPCODESEG,,ATCE,,>
TEMPCODE_SEL =        TEMPCODE-GDT
              ;指向测试任务 TSS 的存储段描述符及选择子
TOTESTTSS    DESC      <TESTTSSLEN-1,TESTTSSSEG,,ATDW,,>
TOTESTTSS_SEL =        TOTESTTSS-GDT
              ;测试任务 TSS 段描述符及选择子
TESTTSS      DESC      <TESTTSSLEN-1,TESTTSSSEG,,AT386TSS,,>
TESTTSS_SEL  =          TESTTSS-GDT

```

;测试任务堆栈段描述符(DPL=1)及选择子

```
TESTSTACK DESC < TESTSTACKLEN - 1, TESTSTACKSEG,, ATDW + DPL1,
D32,>
```

```
TESTSTACK_SEL = TESTSTACK - GDT + RPL1
```

;测试任务代码段描述符(DPL=1)及选择子

```
TESTCODE DESC < 0FFFFH, TESTCODESEG,, ATCE + DPL1, D32,>
```

```
TESTCODE_SEL = TESTCODE - GDT + RPL1
```

```
GNUM = ($ - EFGDT) / (SIZE DESC);需处理基地址的描述符的个数
```

```
;
```

;指向测试任务的任务门

```
TESTTASK GATE < , TESTTSS_SEL,, ATTASKGATE,>
```

```
TEST_SEL = TESTTASK - GDT
```

```
;
```

```
GDTLEN = $ - GDT ;全局描述符表长度
```

```
GDTSEG ENDS ;全局描述符表段定义结束
```

```
;
```

;测试任务的 TSS 段

```
TESTTSSSEG SEGMENT PARA USE16
```

```
TESTTASKSS TSS <> ;TSS 的固定格式部分
```

```
IOMAP LABEL BYTE ;I/O 许可位图
```

```
DB 8 DUP(0FFH) ;端口 00H--3FH
```

```
DB 11110000B ;端口 40H--47H
```

```
DB 3 DUP(0FFH) ;端口 48H--5FH
```

```
DB 11111101B ;端口 60H--67H
```

```
DB 0FFH ;I/O 许可位图结束标志
```

```
TESTTSSLEN = $
```

```
TESTTSSSEG ENDS
```

```
;
```

;测试任务的堆栈段

```
TESTSTACKSEG SEGMENT PARA USE32
```

```
TESTSTACKLEN = 1024
```

```
DB TESTSTACKLEN DUP(0)
```

```
TESTSTACKSEG ENDS
```

```
;
```

;测试任务的代码段

```
TESTCODESEG SEGMENT PARA USE32
```

```
ASSUME CS:TESTCODESEG
```

```
TESTBEGIN PROC FAR
```

```
MOV AL, 0B6H ;使扬声器发出一长声
```

```
OUT 43H, AL
```

```
MOV AL, 2
```

```
OUT 42H, AL
```

```
MOV AL, 34H
```

```
OUT 42H, AL
```



```

        IN      AL,61H
        MOV     AH,AL
        OR      AL,3
        OUT     61H,AL
        MOV     ECX,1234567H
        LOOP    $
        MOV     AL,AH
        OUT     61H,AL
        IRETD
        JMP     TESTBEGIN
TESTBEGIN ENDP
;-----
TESTCODELEN = $
TESTCODESEG ENDS
;-----
;演示任务 TSS 段
DEMOTSSEG   SEGMENT PARA USE16
DEMOTASKSS   TSS    <>
              DB     0FFH                      ;I/O 许可位图结束字节
DEMOTSSLEN   =     $
DEMOTSSEG   ENDS
;-----
;演示任务的堆栈段
DEMOSTACKSEG SEGMENT PARA USE32
DEMOSTACKLEN =     1024
              DB     DEMOSTACKLEN DUP(0)
DEMOSTACKSEG ENDS
;-----
;演示任务的代码段
DEMOCODESEG  SEGMENT  PARA USE32
              ASSUME  CS,DEMOCODESEG
;-----
DEMOBEGIN    PROC  FAR
              MOV     AX,TOTESTTSS_SEL
              MOV     DS,AX
              MOV     EBX,OFFSET TESTTASKSS
              ;把测试任务的入口点,堆栈指针和标志值(含 IOPL)填入测试任务 TSS
              MOV     WORD PTR [EBX]. TRSS,TESTSTACK_SEL
              MOV     DWORD PTR [EBX]. TRESP,TESTSTACKLEN
              MOV     WORD PTR [EBX]. TRCS,TESTCODE_SEL
              MOV     DWORD PTR [EBX]. TREIP,OFFSET TESTBEGIN
              MOV     DWORD PTR [EBX]. TREFLAG,IOPL0
              ;通过任务门调用测试任务
              CALL32 TEST_SEL,0

```

```

                                JUMP32 TEMPCODE_SEL,<OFFSET TODOS>
DEMOBEGIN                      ENDP
;-----
DEMOCODELEN    =          $
DEMOCODESEG    ENDS
;-----
TEMPCODESEG    SEGMENT PARA USE16          ;演示任务的临时代码段
                                ASSUME CS:TEMPCODESEG
;-----
VIRTUAL        PROC    FAR
                                ;置数据段寄存器为空
                                MOV    AX,0
                                MOV    DS,AX
                                MOV    ES,AX
                                MOV    FS,AX
                                MOV    GS,AX
                                ;置堆栈指针
                                MOV    AX,DEMOSTACK_SEL
                                MOV    SS,AX
                                MOV    ESP,DEMOSTACKLEN
                                ;置任务寄存器
                                MOV    AX,DEMOTSS_SEL
                                LTR    AX
                                ;转演示代码段
                                JUMP16 <DEMOCODE_SEL>,<HIGH OFFSET DEMOBEGIN>
TODOS:
                                CLTS
                                MOV    AX,NORMAL_SEL
                                MOV    DS,AX
                                MOV    ES,AX
                                MOV    FS,AX
                                MOV    GS,AX
                                MOV    SS,AX
                                MOV    EAX,CR0
                                AND    AL,11111110B
                                MOV    CR0,EAX
                                JUMP16 <SEG REAL>,<OFFSET REAL>
VIRTUAL        ENDP
;-----
TEMPCODESEG    ENDS
;=====
RDATASEG       SEGMENT PARA USE16          ;实方式数据段
VGDTTR         PDESC <GDTLEN-1,>          ;GDT 伪描述符
SPVAR          DW    ?                     ;用于保存实方式下的 SP
SSVAR          DW    ?                     ;用于保存实方式下的 SS

```

```

RDATASEG      ENDS
;-----
RCODESEG      SEGMENT PARA USE16      ;实方式代码段
               ASSUME    CS:RCODESEG,DS:RDATASEG
;-----
START         PROC
               MOV     AX,RDATASEG
               MOV     DS,AX
               CLD
               CALL    INITGDT          ;初始化全局描述符表 GDT
               LGDT    VGDTR           ;装载 GDTR
               MOV     SSVAR,SS        ;保存堆栈指针
               MOV     SPVAR,SP
               MOV     EAX,CR0
               OR      AL,1
               CLI
               MOV     CR0,EAX
               JUMP16 <TEMPCODE_SEL>,<OFFSET VIRTUAL>
REAL:         MOV     AX,RDATASEG
               MOV     DS,AX
               LSS     SP,DWORD PTR SPVAR ;又回到实方式
               STI
               MOV     AX,4C00H
               INT     21H
START         ENDP
;-----
INITGDT       PROC
               PUSH    DS
               MOV     AX,GDTSEG
               MOV     DS,AX
               MOV     CX,GDNUM
               MOV     SI,OFFSET EFFGDT
INITG:        MOV     AX,[SI].BASEL
               MOVZX   EAX,AX
               SHL     EAX,4
               SHLD    EDX,EAX,16
               MOV     WORD PTR [SI].BASEL,AX
               MOV     BYTE PTR [SI].BASEM,DL
               MOV     BYTE PTR [SI].BASEH,DH
               ADD     SI,SIZE DESC
               LOOP    INITG
               POP     DS
               MOV     BX,16
               MOV     AX,GDTSEG

```

```
MUL    BX
MOV     WORD PTR VGDTR, BASE, AX
MOV     WORD PTR VGDTR, BASE+2, DX
RET
INITGDT ENDP
RCODESEG ENDS
END     START
```

【程序分析】

① 从演示任务切换到测试任务。在从实地址模式切换到保护模式后,就认为进入了演示任务,但 TR 并没有指向演示任务的 TSS。在从演示任务切换到测试任务时,要把演示任务的现场保存到演示任务的 TSS,这就要求 TR 指向演示任务的 TSS。所以,首先使用 LTR 指令把指向演示任务 TSS 描述符的选择子装入 TR。演示任务采用段间调用 CALL,通过任务门 TESTTASK 切换到测试任务。

② 从测试任务切换到演示任务。演示任务通过 IRET 指令切换到测试任务。测试任务的链接字段保存着演示任务的 TSS 中的选择子,取出该选择子,进行任务切换就从测试任务返回到了演示任务。

③ 在测试任务中,由于 I/O 许可位图允许对端口 40H~43H 和 61H 进行读写,因此测试任务能够顺利进行,扬声器发声。如果违反 I/O 许可位图或 I/O 敏感指令将会引起通用保护异常。

习 题

1. 在保护模式下,处理器如何定义一个段? 逻辑地址是如何转换成物理地址的?
2. 在保护模式下,处理器的四个特权级是如何划分的? 哪级最高? 哪级最低?
3. 简述全局描述符表 GDT、局部描述符表 LDT 和中断描述符表 IDT 的作用。在整个系统中,全局描述符表 GDT、局部描述符表 LDT 和中断描述符表 IDT 各有几个?
4. 段描述符高速缓冲寄存器有什么作用?
5. 任务状态段的作用是什么?
6. 在保护模式下,控制转移有哪些情况? 如何实现特权级变换?
7. 简述任务切换过程。
8. 在保护模式下,处理器的中断和异常有哪些? 异常又可分为哪三类?
9. 在保护模式下,处理器是如何转入中断或异常处理程序的?
10. 在保护模式下,处理器如何实现输入/输出保护?
11. 编写一个程序,实现以十六进制的形式显示从内存地址 110000H 开始的 256 个单元的内容。
12. 编写一个保护模式中断程序,实现每隔 1 秒使喇叭发出三声声响。

第 15 章

chapter 15

Windows 汇编语言编程初步

汇编语言和处理器及操作系统都是紧密相关的。随着 DOS 操作系统逐渐地消失,汇编语言程序设计也应从 DOS 下的实地址模式编程过渡到 Windows 下的 32 位保护模式编程。DOS 汇编和 Win32 汇编有相似之处,例如它们有相同的指令系统,寻址方式,这部分内容请参考本书第 3 章。由于篇幅所限,本章只对 Windows 汇编语言应用程序设计做初步介绍。

15.1 Windows 基础

Windows 操作系统工作在保护模式下,本节在第 14 章保护模式的基础上,对 Windows 操作系统中涉及到的 Windows 汇编语言编程的基础知识进行介绍。

1. Windows 的内存管理

Windows 操作系统工作在保护模式下。保护模式是利用分段分页内存管理机制实现虚拟存储器的技术,它在硬盘上建立一个大小为实际内存两倍左右的交换文件用作虚拟内存。实际上 Windows 主要是依赖页式内存管理来调度内存,因为操作系统将每个应用程序的整个 4GB 线性地址空间都作为一个段来处理,这个段总是在内存中(段描述符的 P 位等于 1)。通过页式内存管理,这个段的 4GB(即应用程序),并没有全部装入实际内存。因此,只需在页表中建立映射关系,到了真正被访问时,再将硬盘上的文件调入实际物理内存。

从物理地址空间来看,系统中所有在运行的应用程序都存放在同一个物理地址空间中,而从线性地址空间来看,Windows 操作系统为每一个应用程序建立一个 4GB 的线性空间。因为 Windows 是一个分时的多任务操作系统,CPU 时间被分成一个个的时间片,分配给当前不同应用程序使用。在一个应用程序的时间片内,线性空间中只包含该应用程序的数据段和代码段,操作系统使用的代码和数据(如全局描述符表 GDT,局部描述符表 LDT 与页表等)以及一些共享的代码和数据等,而与该应用程序无关的代码和数据(如其他应用程序的代码和数据)就不能存在于此时的 4GB 线性空间中。Windows 操作系统通过切换页表的内容,可把相同的线性地址映射到不同的物理地址。这样,便实现了应用程序之间的线性地址空间相互隔离的目的。

2. Win32 汇编的内存寻址

在实地址模式下,CPU 对内存采用分段管理,段的大小是 64KB,需要设置段寄存器来指明要访问哪一个段;而在 Windows 系统中,每个应用程序的整个 4GB 线性地址空间都作为一个段。代码段和数据段/堆栈段的空间是统一的,都是 00000000H ~ FFFFFFFFH。在这个 4GB 的地址空间中,一部分用来存放程序,一部分作为数据区,一部分作为堆栈,另外还有一部分被系统使用。这些部分的地址区域是不重合的。

Windows 操作系统不仅预先为要运行的用户应用程序的代码段、数据段和堆栈段设置好描述符,规定这些段的段基址都为 0,段界限都为 FFFFFFFFH,而且程序开始执行时,CS,DS,ES,SS 里存放的选择子已经指向正确的描述符,程序员不需要给这些段寄存器赋值。在整个程序运行期间,程序员也不应该修改这些段寄存器的值。因为在 Windows 系统中,为了保证系统的安全性,不允许用户对描述符表和页表等进行写操作。所以,在编写 Win32 应用程序时,不需要创建段描述符,也不需要创建段描述符表,与实地址模式的汇编相比,Win32 汇编对内存数据的访问更加方便。

3. Windows 下的中断和异常

工作在保护模式下的处理器设置了四个特权级,从高到低分别为特权级 0,1,2,3 级。Windows 操作系统只使用了其中的两个级别,操作系统内核及各种设备驱动程序运行在最高级(0 级),应用程序运行在最低级(3 级)。由于保护模式要求中断和异常处理子程序的特权级比被中断的程序高或相等,因此通常把中断和异常处理子程序放在系统代码中,运行在特权级 0。这样,为了系统的安全,运行在特权级 3 的应用程序是不能像工作在实地址模式下的程序一样,自己来定义中断服务子程序,也不能通过修改中断描述符表来调用系统提供的中断服务子程序。

在 Windows 操作系统中,使用 Windows 提供的应用程序编程接口(Application Programming Interface,API)来代替中断服务子程序提供的系统功能,所以在 Win32 汇编中,INTn 指令失去了存在的意义,在源代码中是看不到 INTn 指令的。Windows 的 API 函数能够被应用程序直接调用,并且它比 DOS 下的中断功能调用具有更丰富的功能。

4. Windows 系统下的 I/O 保护

应用程序运行在特权级 3,如果 CPU 标志寄存器中的 IOPL 设置为 0、1、2,应用程序在读写某个 I/O 端口时,CPU 会检查当前任务的 TSS 状态段中的 I/O 许可位图的对应位,来决定是否对这个 I/O 端口进行读写。如果读写某个 I/O 端口且 I/O 许可位图的对应位为 1,就会引起异常,由操作系统进行异常处理;如果 CPU 标志寄存器中的 IOPL 设置为 3,应用程序就能够访问任何的 I/O 端口而不取决于 I/O 位图。通常,应用程序执行时,对端口是不能直接进行访问的,也不能使用 STI,CLI 中断允许和禁止指令。除非是编写工作在特权级 0 的设备驱动程序。

15.2 Win32 汇编源程序的格式

和 DOS 汇编源程序一样, Win32 汇编源程序也有一定的格式。

15.2.1 源程序结构

一个完整的 Win32 汇编语言源程序结构如下所示:

```
.586
.MODEL FLAT, STDCALL
OPTION CASEMAP:NONE
.DATA
<定义有初始化值的变量>
.....
.DATA?
<定义未初始化值的变量>
.....
.CONST
<定义常量>
.....
.CODE
<标号>
<代码>
.....
END <标号>
```

1. 方式选择伪指令

.586 是一个汇编语言伪指令,含义和 DOS 汇编相同。它通知汇编程序当前的源程序是用哪一种指令集编写的。对于每一种 CPU 有两套功能几乎相同的伪指令: .386/.386P、.486/.486P、.586/.586P。带 P 的指令标明程序中可以用特权级指令,如 MOV CR0,EAX。

2. 内存模式选择伪指令

.MODEL 是用来指定内存模式的伪指令。Windows 操作系统为每一个应用程序建立一个 4GB 的线性空间。所以 Win32 程序只有一种内存模型,那就是 FLAT 平坦模式。代码段和数据段/堆栈段都使用同一个段,内存寻址从 0 到 4GB,没有 64KB 的段大小限制。

定义了.MODEL FLAT,汇编程序自动为各种段寄存器做如下段约定:

```
ASSUME CS:FLAT,DS:FLAT,SS:FLAT,ES:FLAT
```

即 CS,DS,ES,SS 段寄存器都指向同一段 FLAT,FS 和 GS 在 Win32 汇编中默认不用。若要在程序中使用它们,必须自己进行段约定。

STDCALL 告诉汇编程序参数的传递约定。参数的传递约定是指调用函数时参数的压栈顺序(从左到右或从右到左)和由谁恢复堆栈指针(调用者或被调用者)。由于 Windows API 函数调用采用的是 STDCALL 格式,所以 Win32 汇编语言也只能采用 STDCALL 格式。STDCALL 格式的参数传递顺序是从右到左,即最右边的参数最先压栈,由被调用者(子程序)恢复堆栈指针。

3. OPTION 语句

OPTION 语句有很多选项,但对于 Win32 汇编语言程序,这里只介绍一个选项“OPTION CASEMAP:TYPE”。这条语句说明程序中的变量和子程序名是否对大小写敏感,即区分大小写。由于 Windows API 函数是区分大小写的,所以该选项应设置为“OPTION CASEMAP:NONE”。

4. 段定义伪指令

WIN32 中只包含代码段和数据段: DATA 和 CODE。

其中数据段又分为三种,即:“.DATA”,“.DATA?”,“.CONST”。在一个程序里,并不是这三种段都必须有,程序员可以根据需要选择相应的数据段。至于堆栈段,STACK 和实地址模式下相同,可以不必考虑,因为系统会自动为应用程序分配一个向低地址扩展的足够大的段作为堆栈段。

.DATA 定义已初始化的变量。这些变量的值在程序的执行中可以被更改。

.DATA? 定义未初始化的变量。如果仅想预先分配一些内存但并不想指定初始值,可以使用未初始化的变量。使用未初始化的数据的优点是,它不占据可执行文件的大小,仅仅是在装载可执行文件时分配所需字节的内存空间。

.CONST 定义常量,这些常量在程序运行过程中是不能更改的。如果在程序中出现对.CONST 定义的常量进行写操作,会引起异常。

.CODE 定义代码段,所有的指令都必须写在代码段,因为只有代码段才有可执行属性。对于运行在特权级 3 的应用程序来说,.CODE 段是不可写的。

5. 汇编结束语句

格式:

标号:

END 标号

和 DOS 汇编相同,它通知汇编程序,源程序到此结束,标号所对应的指令是程序的启动指令。程序的所有可执行代码必须在这两者之间。

15.2.2 Windows API 函数的应用

下面给出一个完整的 Win32 汇编语言经典的源程序的例子,该程序运行时将弹出一

个消息框并显示字符串“Hello,你好!”。程序的执行结果如图 15-1 所示。

【例 15.2.1】

```
.586
.MODEL FLAT,STDCALL
OPTION CASEMAP:NONE
INCLUDE WINDOWS.INC
INCLUDE KERNEL32.INC
INCLUDELIB KERNEL32.LIB
INCLUDE USER32.INC
INCLUDELIB USER32.LIB

.DATA
MsgBoxCaption DB 'Example of win32',0    ;消息框标题显示字符串
MsgBoxText    DB 'Hello,你好!',0        ;消息框内显示字符串

.CODE
START:
INVOKE MessageBox, NULL, addr MsgBoxText, addr MsgBoxCaption, MB_OK
INVOKE ExitProcess, NULL
END START
```



图 15-1 程序的执行结果

1. Windows API 函数

从例 15.2.1 可见,弹出字符串显示对话框调用了函数 MessageBox,若退出程序执行,则调用函数 ExitProcess。函数 MessageBox 和 ExitProcess 都是 Windows API 函数。

在前面一节介绍过,在 Windows 操作系统中,使用 API 函数替代了 DOS 的软中断 INTn。API 是一个函数集合,函数的大部分被包含在几个动态链接库(Dynamic Link Library,DLL)中。所谓动态链接库,是指这些 API 的代码本身并不包含在 Windows 可执行文件中,而是当要使用时才被加载。Win32 API 的核心由 3 个 DLL 提供。kernel32.dll 中的函数主要处理内存管理和进程调度;user32.dll 中的函数主要控制用户界面,包括创建窗口和传递消息;gdi32.dll 中的函数则负责图形方面的操作。

2. 函数的声明

对于程序中所有要用到的 API 函数,在程序的开始部分都必须预先声明。包括函数的名称、参数的类型等。声明函数的格式是:

函数名 PROTO [调用规则]: [参数 1]: 数据类型,[参数 2]: 数据类型,……

其中,[调用规则]是可选项,如果不写,则使用.MODEL 语句中指定的调用规则 STDCALL。

后面的参数列表中的参数名称可以省略,参数类型对于 Win32 汇编来说,只存在 DWORD 类型。

例如在汇编程序中使用的 MessageBox 函数声明如下:


```
MessageBox PROTO hWnd:DWORD,lpText: DWORD ,lpCaption: DWORD,uType: DWORD
```

上述函数声明说明了 MessageBox 有 4 个参数,它们分别是 HWND 类型的窗口句柄(hWnd),LPCTSTR 类型的要显示的字符串地址(lpText)和标题字符串地址(lpCaption),还有 UINT 类型的消息框类型(uType)。在 Microsoft 发布的 Microsoft Win32 Programmer's Reference 中定义了常用 API 的参数和函数声明。

3. INCLUDE 语句

如果要将程序中所有的函数都予以声明,显然十分麻烦。因此为了简化操作,可将所有的声明预先放在一个文件中,用 INCLUDE 语句包含进来。例 15.2.1 用到了两个 API 函数: MessageBox 和 ExitProcess,它们分别在 user.dll 和 kernel32.dll 中,所对应的 API 函数声明文件 user32.inc 和 kernel.inc。在源程序中用 INCLUDE 语句包含进来就可以了。

在 Win32 应用程序中,一般都还应该有一条包含语句:

```
INCLUDE WINDOWS.INC
```

Windows.inc 文件包含了几乎所有 API 函数的参数常量和数据结构定义。例如 MessageBox 的 uType 参数中使用的常量 MB_OK 就定义在这个文件中。

uType——定义对话框的类型,这个参数可以是以下标志的集合:

要定义消息框上的显示按钮,用下面的某一个标志:

MB_ABORTRETRYIGNORE——消息框有三个按钮:“终止”、“重试”和“忽略”。

MB_HELP——消息框上显示一个“帮助”按钮,按下后发送 WM_HELP 消息。

MB_OK——消息框上显示一个“确定”按钮,这是默认值。

MB_OKCANCEL——消息框上显示两个按钮:“确定”和“取消”。

MB_RETRYCANCEL——消息框上显示两个按钮:“重试”和“忽略”。

MB_YESNO——消息框上显示两个按钮:“是”和“否”。

MB_YESNOCANCEL——消息框上显示三个按钮:“是”、“否”和“取消”。

要在消息框中显示图标,用下面的某一个标志:

MB_ICONWARNING —— 显示惊叹号图标。

MB_ICONINFORMATION —— 显示消息图标。

MB_ICONASTERISK —— 显示危险图标。

MB_ICONQUESTION —— 显示问号图标。

MB_ICONSTOP —— 显示停止图标。

4. INCLUDELIB 语句

不同类的 API 函数存放在不同的动态链接库 DLL 中,为了让链接程序快速地搜索到 API 函数在哪个 DLL 库,Win32 还定义了一种库文件,称为导入库文件。一个 DLL 库对应一个导入库。INCLUDELIB 语句就用于指定链接时所用的导入库,用于通知链



接程序在哪个 DLL 库中去找链接的 API 函数。例如, user32. dll 对应的导入库是 user32. lib, 程序中应加一条语句: INCLUDELIB USER32. LIB。

5. API 函数的调用

在汇编语言中, 应该用 CALL 指令来调用 MessageBox 函数。本书在第 5 章中介绍过, 汇编语言中子程序的调用是不能通过形参和实参一一对应的方法来传递参数的。如果子程序传递的参数较多, 则调用子程序就比较麻烦。而 Win32 API 函数与 DOS 系统功能调用相比, 参数传递量往往很大, 这样的调用显然十分容易出错。好在 Microsoft 在 MASM 中提供了一个伪指令实现了这个功能, 那就是 INVOKE 伪指令, 它的格式是:

INVOKE 函数名[, 参数 1][, 参数 2]……

语句中的参数是实参, 可以是数值表达式, 寄存器或“ADDR 变量名”, 其中“ADDR 变量名”传送的是变量地址。

INVOKE 在汇编的时候, 展开成下面的指令:

```
PUSH uType  
PUSH lpCaption  
PUSH lpText  
PUSH hWnd  
CALL MessageBox
```

6. API 函数的返回值

有的 API 函数有返回值, 如 MessageBox 定义的返回值是 INT 类型的数, 返回值的类型对汇编程序来说也只有 DWORD 一种类型, 它永远放在 EAX 中。如果要返回的内容不是一个 EAX 所能容纳的, Win32 API 采用的方法一般是返回一个指针, 或者在调用参数中提供一个缓冲区地址, 把数据直接返回到缓冲区中去。

15.3 Win32 汇编可执行文件的生成

和 DOS 汇编源程序相同, Win32 汇编源程序也必须经过汇编和链接后才能生成可执行文件。但除了源程序外, 资源(包括菜单、图标、对话框、各种控件)文件也是 Win32 应用程序的重要组成部分。因此 Win32 汇编软件的开发可分为源程序开发和资源开发两部分。其中, 源程序的开发过程和 DOS 源程序相同, asm 源程序经汇编程序汇编成 obj 目标程序; 资源文件的“源文件”是以 rc 为扩展名的脚本文件, 由资源编译程序编译成以 res 为扩展名的二进制资源文件; 最后由链接程序将它们链接成可执行文件。

图 15-2 给出了 Win32 汇编可执行文件的生成过程。其中资源开发部分不是每一个 Win32 应用程序都必须的。

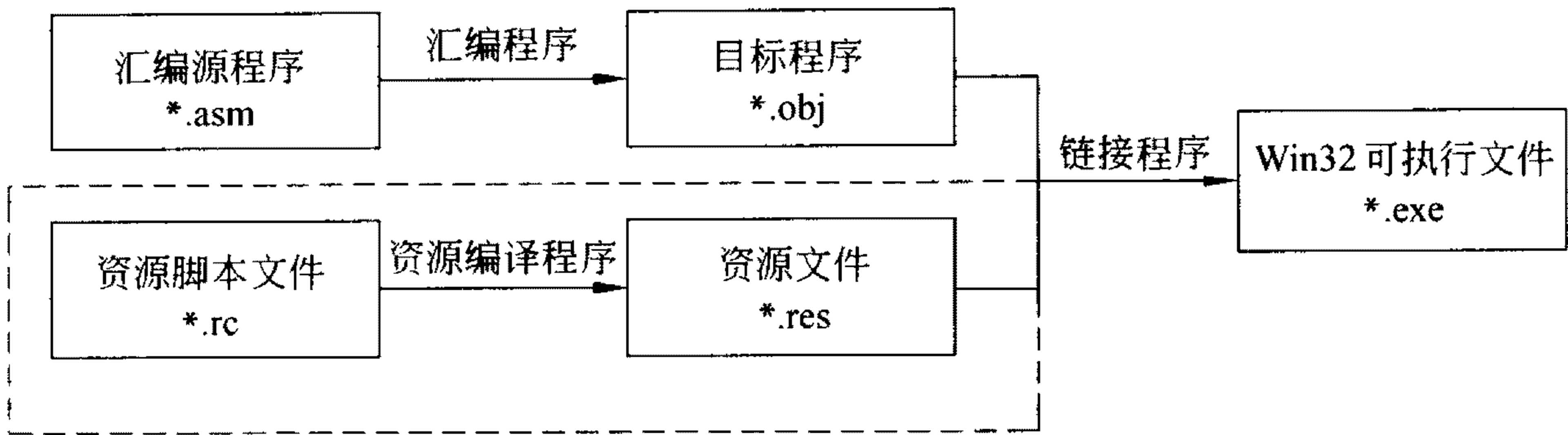


图 15-2 Win32 汇编可执行文件的生成

15.3.1 汇编和链接

1. 汇编

目前汇编语言汇编软件有很多,常用的是 Microsoft 公司的 MASM 和 Borland 公司的 TASM。对 Win32 汇编来说,MASM 的使用最为方便,它支持@@标号,可用 invoke 调用子程序,支持局部变量并有高级语法。因此下面以 MASM6.14 为例介绍它的用法。

MASM 汇编器的命令行用法为:

ml [/选项] 汇编源文件列表

ml 在 Win32 汇编中常用的选项如表 15-1 所示。

表 15-1 ml 的常用选项

选 项	简 介
/c(常用)	仅进行汇编,不自动进行链接
/coff(必用)	产生的 obj 文件格式为 COFF 格式
/Cp	源代码区分大小写
/Fo filename	指定输出的 obj 文件名
/Fe filename	指定链接后输出的 exe 文件名
/I pathname	指定 include 文件的路径
/link 选项	指定链接时使用的选项
/Zi(调试程序常用)	增加符号调试信息

2. 链接

用 ml.exe 编译的 COFF 格式的 obj 文件可以用 Link.exe 链接成可执行 PE 文件,Microsoft 的 link.exe 有两个系列的版本,用于链接 DOS 程序的链接器为 Segmented Executable Linker;可以链接 Win32 PE 文件的链接器为 Incremental Linker,这里指的是 Incremental Linker 的用法。

link 的命令行使用方法为:

link [选项] [文件列表]

命令行参数中的文件列表用来列出所有需要链接到可执行文件中的模块,可以指定多个 obj 文件、res 资源文件以及导入库文件。link 的选项很多,常用的选项如表 15-2 所示。

表 15-2 link 的常用选项

选 项	简 介
/DEBUG(调试程序常用)	在 PE 文件中加入调试信息
/DRIVER: 类型	链接 Windows NT 的 WDM 驱动程序时用,类型可以是 WDM 或者 UPONLY
/DLL	链接动态链接库文件时用
/DEF: 文件名	编写链接库文件时使用的 def 文件名,用来指定要导出的函数列表
/IMPLIB: 文件名	当链接有导出函数的文件时(如 DLL)要建立的导入库名
/LIBPATH: 路径	指定库文件的目录
/OUT: 文件名	指定输出文件名,默认的扩展名是 .exe,如果要生成其他文件名,如屏幕保护 *.scr 等,则在这里指定
/STACK: 尺寸	设定堆栈尺寸
/SUBSYSTEM: 系统名	指定程序运行的操作系统,可以是 NATIVE, WINDOWS, CONSOLE, WINDOWSCE 和 POSIX 等

3. 资源文件的生成

资源文件包括菜单、对话框、字符串、图标、位图资源等,在链接时,链接程序将资源加入到可执行文件中去。资源是由一些脚本文件构成的,它可以用普通的文件编辑器来编辑,也可以用更适合编写资源文件的所见即所得的资源编辑器来编辑。如 Visual C++ 资源编辑器。

资源编辑器在对资源编辑完成后,可以用两种格式来保存。

- ① 将该资源文件保存为.rc 格式的文件,然后再用资源编译程序 rc.exe,将.rc 文件编译成.res 文件。
- ② 直接将该资源文件保存为.res 格式的文件,这样,便可以在链接时直接将.res 文件链接到可执行文件。

4. 汇编链接步骤

以例 15.2.1 为例,对它进行汇编链接,最后运行。

- (1) 汇编源程序 hello.asm ml /c /coff hello.asm
- (2) 链接目标程序 hello.obj link /subsystem:windows hello.obj
- (3) 运行可执行程序 hello.exe hello.exe

注意: 如果该程序有资源文件 hello.res,则应该用 link 将 hello.obj 和 hello.res 链接成 hello.exe。

15.3.2 调试 Win32 汇编程序

Win32 汇编语言不像高级语言那样具有完善的集成开发环境,如 Microsoft Visual C++,在集成环境中,不仅可以实现编辑、编译、链接等工作,还具有调试功能;而 Win32 汇编语言程序开发,则需要额外选择调试工具。

Numega 公司的 SoftICE 是目前比较流行的一种调试工具,用它来调试 Win32 汇编程序十分方便。SoftICE 既可用于调试各种运行于特权级 3 的应用程序,也可用于调试运行于特权级 0 的设备驱动程序;甚至可以调试 Windows 操作系统本身。由于 SoftICE 运行于硬件与 Windows 之间,因此 SoftICE 的用户界面与 Windows 系统相互独立。当按下其激活热键[Ctrl+D]后,Windows 桌面隐藏,SoftICE 弹出,这时可对用户程序进行调试。

注意,如果要对 Win32 源程序进行调试,可执行文件(.exe)中必须含有调试信息。因此必须使用带/Zi 选项的 ml.exe 对源程序进行汇编;并用带/DEBUG 选项的 link.exe 对目标程序进行链接。

15.4 Win32 汇编基本语法

本节着重介绍 Win32 汇编语言的基本语法,和 DOS 汇编相同部分不再介绍,请参考本书第 4 章。

15.4.1 标号和变量

1. 标号

在 DOS 汇编中,标号的作用域是整个程序,在整个程序中是唯一的。但在 Win32 汇编使用的高版本汇编中,标号的作用域是当前的子程序。在同一个子程序中的标号不能同名,但在不同的子程序中可以有相同名称的标号,即不用转移指令从一个子程序能跳到另一个子程序。

2. 变量

和 DOS 汇编相比,Win32 汇编中变量的类型很多,如表 15-3 所示。而且根据变量的作用域可分为全局变量和局部变量。

表 15-3 变量的类型

名 称	表示方式	缩 写	长度(字节)
字节	BYTE	DB	1
字	WORD	DW	2
双字(DOUBLEWORD)	DWORD	DD	4



续表

名 称	表示方式	缩 写	长度(字节)
三字(FARWORD)	FWORD	DF	6
四字(QUADWORD)	QWORD	DQ	8
十字节 BCD 码(TENBYTE)	TBYTE	DT	10
有符号字节(SIGNBYTE)	SBYTE		1
有符号字(SIGNWORD)	SWORD		2
有符号双字(SIGNDWORD)	SDWORD		4
单精度浮点数	REAL4		4
双精度浮点数	REAL8		8
10 字节浮点数	REAL10		10

(1) 全局变量

和 DOS 汇编相同,可用数据定义伪指令在 .DATA 或 .DATA? 段定义全局变量。全局变量的作用域是整个程序。

(2) 局部变量

局部变量的作用域是当前的子程序。定义的格式是:

LOCAL 变量 1[重复数量 1][: 类型],变量 1[重复数量 2][: 类型]……

LOCAL 伪指令必须紧跟在子程序的伪指令 PROC 之后,其他指令之前。如果定义的变量是 DWORD 类型,可以省略类型。局部变量不能和全局变量同名。

例:

```
LOCAL VAR1:WORD      //定义了一个字局部变量
LOCAL VAR2            //定义了一个双字局部变量
LOCAL VAR3[10]:BYTE  //定义了有 10 字节长的局部变量
```

例: 在程序中使用局部变量

```
N1 PROC
    LOCAL VAR1:DWORD,VAR2:WORD
    LOCAL VAR3:BYTE
    MOV  EAX,VAR1
    MOV  AX,VAR2
    MOV  AL,VAR3
    RET
N1 ENDP
```

局部变量是被存放在堆栈空间的。CPU 进入子程序的时候就会根据该子程序里面的局部变量所需空间的大小,在堆栈中留出相应大小的空间供局部变量使用。因此局部变量无法初始化,只能在程序里用指令给它们赋值。

3. 获取变量地址

获取变量地址的操作对于全局变量和局部变量是不同的。

获取全局变量地址的方法和 DOS 汇编相同,可使用 OFFSET 运算符。

例:

```
MOV BX,OFFSET 变量名
```

对于局部变量,MASM 对此有一个专用的伪操作符 ADDR。

格式:

```
ADDR 局部变量名和全局变量名
```

例:

```
INVOKE MessageBox, NULL, ADDR MsgBoxText, ADDR MsgBoxCaption, MB_OK
```

当 ADDR 后面跟全局变量名的时候,用法和 OFFSET 是相同的。注意 ADDR 伪操作符只能在 INVOKE 的参数中使用,不能用在类似于下列的指令中:

```
MOV EAX,ADDR 局部变量名 ;该指令错误
```

15.4.2 结构

在 MASM 下的 Win32 汇编语言程序里面,可以使用关键字 STRUCT 来定义一个所需要的数据类型。

1. 结构的定义

格式:

```
结构名 STRUCT
```

```
    字段 1  类型  ?
```

```
    字段 2  类型  ?
```

```
    .....
```

```
结构名 ENDS
```

结构中间的每一个字段可以是字节、字、双字、字符串或所有可能的数据类型。结构的定义也可以嵌套。

例:定义一个名为 STUDENT 的结构,该结构有 3 个字段。

```
STUDENT STRUCT
```

```
    NUM BYTE  ?
```

```
    SEX BYTE  ?
```

```
    RECORD WORD  ?
```

```
STUDENT ENDS
```

2. 结构变量的定义

定义了结构以后,就可以在程序里定义对应的结构变量。结构变量的定义格式

如下:

变量名 结构名 <> 格式一

或

变量名 结构名 <VAR1,VAR2,……> 格式二

格式一定义的变量是未初始化的。格式二在定义变量的同时指定结构中各字段的初始值,各字段的初始值用逗号隔开。

例:定义结构变量

```
STU1 STUDENT <>
```

```
STU2 STUDENT <10,0,100>
```

3. 结构变量的访问

在汇编中,结构变量的访问方法有好几种。

① 以上面的定义为例,如果要使用 STU2 中的 RECORD 字段,最直接的办法是:

```
MOV AX,STU2.RECORD
```

它表示把 RECORD 字段的值放入 AX 中去。

② 在实际使用中,常常有使用指针存取数据结构的情况,如果使用 ESI 寄存器做指针寻址,可以使用下列语句完成同样的功能:

```
MOV ESI,OFFSET STU2
```

```
MOV AX,[ESI + STUDENT.RECORD]
```

注意:第二句是[ESI + STUDENT.RECORD]而不是[ESI + STU2.RECORD]。

③ 如果要对一个数据结构中的大量字段进行操作,MASM 还有一个比较简便的用法,可以用 ASSUME 伪指令把寄存器预先定义为结构指针,再进行操作:

```
MOV ESI,OFFSET STU2
```

```
ASSUME ESI:PTR STUDENT
```

```
MOV EAX,[ESI].RECORD
```

```
...
```

```
ASSUME ESI:NOTHING
```

这样,程序的可读性比较好。注意:在不再使用 ESI 寄存器做指针的时候要用 ASSUME ESI:NOTHING 取消定义。

15.4.3 子程序

Win32 汇编中的子程序都采用堆栈来传递参数,这样可以用 INVOKE 伪指令来进行调用和语法检查。

1. 子程序的定义

格式:

```
子程序名 PROC [语言类型][可视区域][USES 寄存器列表][,参数:类型]...[VARARG]
            LOCAL 局部变量列表
            .....
            RET
子程序名 ENDP
```

PROC 和 ENDP 伪指令定义了子程序开始和结束的位置,Win32 汇编子程序的属性意义与 DOS 汇编不同。各属性意义如下:

(1) 语言类型

表示参数的使用方式和堆栈平衡的方式,可以是 STDCALL,C,SYSCALL,BASIC,FORTTRAN 和 PASCAL,如果忽略,则使用程序头部 .MODEL 定义的值。Win32 约定的类型是 STDCALL,参数传递顺序是从右到左,由子程序恢复堆栈指针,所以在程序中调用子程序或系统 API 后,不必自己来平衡堆栈。

(2) 可视区域

可以是 PRIVATE,PUBLIC 和 EXPORT。PRIVATE 表示子程序只对本模块可见;PUBLIC 表示子程序对所有的模块可见;EXPORT 表示子程序是导出的函数,当编写 DLL,要将某个函数导出的时候可以这样使用。默认的设置是 PUBLIC。

(3) USES 寄存器列表

表示 CPU 在进入子程序后自动执行 PUSH 这些寄存器的指令,在 RET 子程序返回前自动执行 POP 指令,用于保护现场。程序员也可以自己在子程序开头和结尾用 PUSHAD 和 POPAD 指令一次保存和恢复所有的寄存器。

(4) 参数和类型

参数指参数的名称,在定义参数名的时候不能跟全局变量和子程序中的局部变量重名。对于类型,由于 Win32 中的参数类型只有 32 位(DWORD)一种类型,所以可以省略。

(5) VARARG

表示在已确定的参数后还可以跟多个数量不确定的参数,在 Win32 汇编中惟一使用 VARARG 的 API 就是 wsprintf,类似于 C 语言中的 printf,其参数的个数取决于要显示的字符串中指定的变量个数。

2. 子程序的声明和调用

完成了子程序的定义之后,可以用 CALL 指令或更方便的 INVOKE 伪指令来调用子程序。

如果是用 CALL 指令来调用子程序,必须自己来完成参数的传递,参数传递的方法请参见第 5 章。

如果是用 INVOKE 指令来调用,可以通过形参和实参一一对应的方法来传递参数。



而且要注意的是,如果子程序的定义位于调用指令之后,则必须在程序头部同 PROTO 伪指令预先给出子程序声明。如果子程序的定义位于调用指令之前,则不必进行子程序的声明。PROTO 的用法见 15.2.2 节。

【例 15.4.1】 假设 $N1=1122H$, $N2=3344H$, $N3=5566H$, 利用子程序完成三个数相加

【程序清单】

```
.586
.MODEL FLAT,STDCALL
OPTION CASEMAP:NONE
INCLUDE KERNEL32.INC
INCLUDELIB KERNEL32.LIB
INCLUDE WINDOWS.INC
COMPUTE PROTO PARA1:DWORD           ;计算子程序声明
.DATA
N1 DW 1122H
N2 DW 3344H
N3 DW 5566H
.DATA?
SUM DW ?                             ;计算结果
.CODE
START:
INVOKE COMPUTE, ADDR N1             ;调用计算子程序,将数据存放的起始地址作为传递的参数
MOV SUM, AX                         ;保存计算结果
INVOKE ExitProcess, NULL            ;结束执行程序
COMPUTE    PROC USES EAX EBX, PARA1:DWORD
    MOV EBX, PARA1
    MOV EAX, 0                      ;求和寄存器清 0
    MOV AX, WORD PTR [EBX]          ;AX=N1
    ADD AX, WORD PTR [EBX+2]         ;AX=N1+N2
    ADD AX, WORD PTR [EBX+4]         ;AX=N1+N2+N3
    RET
COMPUTE    ENDP
END START
```

15.4.4 高级语法

以前高级语言和汇编的最大差别就是条件测试、分支和循环等高级语法。高级语言中,程序员可以方便地用类似于 IF, CASE, LOOP 和 WHILE 等语句来构成程序的结构流程,不仅条理清楚,而且可维护性好。而汇编程序只能用 CMP、TEST 和众多的条件转移指令来完成,使得汇编源程序可读性差,编程复杂。

现在 Win32 的高版本 MASM 中新引入了一系列的伪指令,涉及条件测试、分支和循环语句。利用它们,汇编语言有了和高级语言一样的结构,配合局部变量和调用参数,为

使用 Win32 汇编编写大规模的 Windows 应用程序奠定了基础。

1. 条件测试表达式

在所有的分支和循环语句首先要进行条件测试,也就是判断一个表达式的结果是“真”还是“假”,来决定程序的走向。表达式中往往有用来做比较和计算的操作符。

条件测试的基本表达式是:
寄存器或变量 操作符 操作数

两个以上的表达式可以用逻辑运算符连接:
(表达式 1)逻辑运算符(表达式 2) 逻辑运算符(表达式 3)……

另外又增加了 CPU 标志寄存器一些标志位的状态,它们本身相当于一个表达式。
允许的操作符、逻辑运算符和标志位状态如表 15-4 所示。

表 15-4 操作符、逻辑运算符和标志位状态

操作符	操作	逻辑运算符	操作	标志位状态	操作
==	等于	!	逻辑取反	CARRY?	CF=1 为真
!=	不等于	&&	逻辑与	OVERFLOW?	OF=1 为真
>	大于		逻辑或	PARITY?	PF=1 为真
>=	大于等于			SIGN?	SF=1 为真
<	小于			ZERO?	ZF=1 为真
<=	小于等于				
&	位测试				

注意:

- ① 条件测试语句有几个限制,首先表达式的左边只能是变量或寄存器,不能为常数;其次表达式的两边不能同时为变量,但可以同时是寄存器。
- ② 与 CMP 和 TEST 指令相同,条件测试伪指令并不会改变被测试的变量或寄存器的值。
- ③ 若两个数用关系运算符>、>=、<和<=进行大小比较时,汇编程序默认它们为无符号数比较。如果程序员认为这两个数是有符号数,进行的是有符号数比较,可在左边的操作数前加上“SDWORD PTR”,“SWORD PTR”“SBYTE PTR”运算符。

例如:

```
X==10           ;X 等于 10 为真
EAX!=0          ;EAX 不等于 0 为真
SBYTE PTR AL>=8 ;AL 大于等于 8 时为真,AL 为有符号数
(X>=100)&&ECX    ;X 大于等于 100 且 ECX 为非零时为真
(Y&80H)||! EAX   ;Y 和 80H 进行“与”操作后非零或 EAX 取反后非零时为真
(EAX==EBX)&&ZERO? ;EAX 等于 EBX 且 Z 标志=1 为真
```

2. 分支语句

分支语句用来根据条件表达式测试的真假执行不同的代码,分支语句的语法如下:


```

ENDIF

```

【程序清单】

MOV EBX,OFFSET MSG3

```
.ENDIF
    INVOKE MessageBox, NULL, EBX, ADDR MsgBoxCaption, MB_OK
    INVOKE ExitProcess, NULL
END START
```

上述例子利用了条件分支高级语法来实现,与本书例 5.5.2 相比,该程序可读性好,容易理解。

3. 循环语句

循环是重复执行的一组指令,循环伪指令可以根据条件表达式的真假来控制循环是否继续,也可以在循环体中直接退出。与高级语言相同,循环结构的语法也有两种。

(1) 循环结构语法一

```
.WHILE 条件测试表达式
    指令
    [. BREAK [. IF 退出条件]]
    [. CONTINUE]
.ENDW
```

. WHILE/. ENDW 循环首先判断条件测试表达式,如果结果是“真”,则执行循环体内的指令,结束后再回到 . WHILE 处判断表达式,如此往复,一直到表达式结果为“假”为止。. WHILE/. ENDW 指令有可能一遍也不会执行到循环体内的指令,因为如果第一次判断表达式时就遇到结果为“假”的情况,那么就直接退出循环。

(2) 循环结构语法二

```
.REPEAT
    指令
    [. BREAK [. IF 退出条件]]
    [. CONTINUE]
```

. UNTIL 条件测试表达式 (或. UNTILCXZ [条件测试表达式])

. REPEAT/. UNTIL 循环首先执行一遍循环体内的指令,然后再判断条件测试表达式,如果结果为“真”的话,就退出循环,如果为“假”,则返回 . REPEAT 处继续循环,可以看出,. REPEAT/. UNTIL 不管表达式的值如何,至少会执行一遍循环体内的指令。

也可以把条件表达式直接设置为固定值,这样就可以构建一个无限循环,对于 . WHILE/. END 直接使用 TRUE,对 . REPEAT/. UNTIL 直接使用 FALSE 来当表达式就是如此,这种情况下,可以使用 . BREAK 伪指令强制退出循环,如果 . BREAK 伪指令后面跟一个 . IF 测试伪指令的话,那么当退出条件为“真”时才执行 . BREAK 伪指令。

在循环体中也可以用 . CONTINUE 伪指令忽略以后的指令,遇到 . CONTINUE 伪指令时,不管下面还有没有其他循环体中的指令,都会直接回到循环头部开始执行。

【例 15.4.3】 假设从 BUF 单元开始为一个 ASCII 码字符串,找出其中的最大数送屏幕显示。

【程序清单】

```
.586
.MODEL FLAT,STDCALL
OPTION CASEMAP:NONE
INCLUDE KERNEL32.INC
INCLUDE WINDOWS.INC
INCLUDELIB KERNEL32.LIB
INCLUDE USER32.INC
INCLUDELIB USER32.LIB
.DATA
BUF    DB 'QWERTYUIOP123'
COUNT EQU $-BUF
MAX    DB 'Max=',?,0
MsgBoxCaption    DB "Example of win32",0
.CODE
START:
    MOV ECX,0
    MOV EBX,OFFSET BUF           ;字符串首址偏移->EBX
    MOV AL,0                     ;最小数->AL
    .WHILE ECX<COUNT            ;循环
        MOV DL,[EBX]
        .IF(DL>AL)               ;比较
            MOV AL,DL            ;大数->AL
        .ENDIF
        INC EBX                  ;调整字符串首址偏移
        INC ECX
    .ENDW
    MOV MAX+4,AL                 ;保存最大值
    INVOKE MessageBox, NULL, ADDR MAX, ADDR MsgBoxCaption, MB_OK
    INVOKE ExitProcess, NULL
END START
```

15.5 创建 Windows 下的窗口程序

在 Windows 这样一个多任务操作系统中,一个需要和用户交互的应用程序可以通过一个或多个窗口来达到与用户交换信息的目的。窗口是 Windows 操作系统下应用程序的基础。本节编写一个以标准的窗口为界面的程序。

15.5.1 窗口程序的运行过程

DOS 程序设计采取的是一种顺序化的,按过程驱动的程序设计方法;而窗口程序采用的是消息驱动程序的设计方法。所有的用户操作,如用户按键、鼠标移动、选择菜单和

拖动窗口等都是通过消息来传给应用程序的。应用程序中由窗口过程接收消息并处理。所以窗口过程是整个应用程序的重点。窗口过程的运行过程如下：

① 当用户进行操作时，Windows 将会以消息的形式记录下这些操作，送到系统的消息队列中。

② 检查该消息发生在哪个应用程序的窗口范围，将这个消息送到该应用程序自己的消息队列中。

③ 应用程序会不断地执行消息循环过程，当执行到 GetMessage 函数时，该函数会从应用程序消息队列中取出一条消息到应用程序。

④ 应用程序用 TranslateMessage 函数对这条消息进行预处理，再调用 DispatchMessage 函数，将这条消息的有关信息作为参数传递给窗口过程，并回调窗口过程对消息进行处理。窗口过程对消息处理结束，又返回到 DispatchMessage 函数代码段中。执行完 DispatchMessage 函数后，又回到应用程序的消息循环中，继续下一次消息循环。

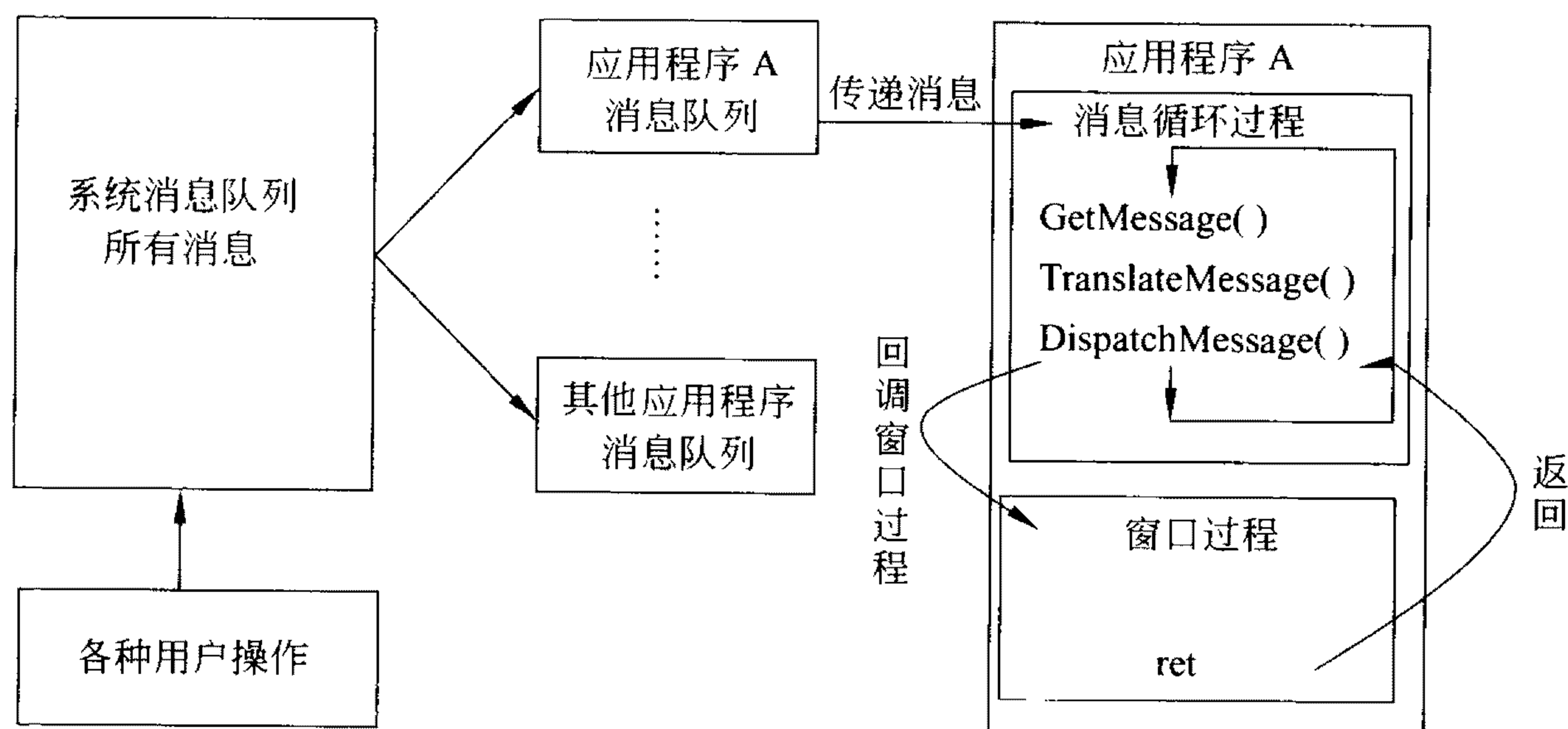


图 15-3 窗口程序的运行过程

15.5.2 窗口程序示例

在 Windows 操作系统下创建并显示一个窗口程序的编程步骤为：

- ① 调用 GetModuleHandle 函数获得应用程序的句柄。
- ② 可以根据需要从命令行得到参数。
- ③ 如果不是使用 Windows 预定义的窗口类，如 MessageBox 或 dialog box，必须先填写用户自定义窗口注册类的结构（WNDCLASSEX）的变量参数，再调用 RegisterClassEx 函数注册窗口类。
- ④ 如果想程序运行后，立即在桌面显示窗口，调用 ShowWindow 函数显示窗口。
- ⑤ 调用 UpdateWindows 函数刷新窗口客户区。
- ⑥ 进入消息循环。
- ⑦ 如果有消息到达，则由该窗口的窗口回调函数，即用户写的窗口过程，对消息进行处理。

【例 15.5.1】 设计一个基本的 Win32 窗口程序,该程序显示一个标准 Windows 窗口,在窗口范围内单击鼠标左键,将弹出一个消息框并显示字符串“Hello,你好!”。程序的执行结果应如图 15-4 所示。

【程序清单】

```
.586
.MODEL FLAT, STDCALL
OPTION CASEMAP:NONE
WinMain PROTO : DWORD, :, DWORD, :, DWORD, :, DWORD
INCLUDE \MASM32\INCLUDE\WINDOWS.INC
INCLUDE \MASM32\INCLUDE\USER32.INC
INCLUDE \MASM32\INCLUDE\KERNEL32.INC
INCLUDE \MASM32\INCLUDE\GDI32.INC
INCLUDELIB \MASM32\LIB\USER32.LIB
INCLUDELIB \MASM32\LIB\KERNEL32.LIB
INCLUDELIB \MASM32\LIB\GDI32.LIB
.DATA
ClassName      db 'SimpleWinClass',0           ;定义用户的窗口类名
AppName        db 'Window Example',0           ;窗口标题
szCaption       db 'Example of win32',0         ;消息框标题字符串
szText          db 'Hello,你好!',0             ;消息框内显示字符串
.DATA?
hInstance       HINSTANCE ?                   ;保存应用程序句柄
CommandLine     LPSTR ?                       ;保存命令行参数
.CODE
START:
    INVOKE GetModuleHandle, NULL               ;得到应用程序句柄
    MOV     hInstance,EAX                      ;保存应用程序句柄
    INVOKE GetCommandLine                      ;得到命令行参数
    MOV     CommandLine,EAX                   ;保存命令行参数
    INVOKE WinMain,hInstance,NULL,CommandLine,SW_SHOWDEFAULT
    INVOKE ExitProcess,EAX                    ;结束程序执行

WinMain PROC  hInst: HINSTANCE, hPrevInst: HINSTANCE, CmdLine: LPSTR,
CmdShow:DWORD
    LOCAL wc:WNDCLASSEX                      ;窗口注册类结构变量
    LOCAL msg:MSG                            ;消息结构变量
    LOCAL hwnd:HWND                          ;本窗口句柄
;-----
; 注册窗口类
;-----
    MOV     wc.cbSize,SIZEOF WNDCLASSEX       ;结构大小
    MOV     wc.style,CS_HREDRAW or CS_VREDRAW ;窗口外型风格
    MOV     wc.lpfnWndProc,OFFSET WndProc     ;设置窗口消息处理过程
```



```

MOV  wc.cbClsExtra, NULL
MOV  wc.cbWndExtra, NULL
PUSH hInst
POP  wc.hInstance           ;设置应用程序句柄
MOV  wc.hbrBackground, COLOR_WINDOW+1 ;设置窗口背景色
MOV  wc.lpszMenuName, NULL
MOV  wc.lpszClassName, OFFSET ClassName ;设置窗口类名
INVOKE LoadIcon, NULL, IDI_APPLICATION
MOV  wc.hIcon, eax          ;设置窗口程序的图标
MOV  wc.hIconSm, eax        ;设置窗口标题栏中的小图标
INVOKE LoadCursor, NULL, IDC_ARROW
MOV  wc.hCursor, eax        ;设置在该窗口显示的光标形状
INVOKE RegisterClassEx, addr wc ;注册用户定义窗口类
;-----
;  创建窗口
;-----
INVOKE CreateWindowEx, NULL, addr ClassName, ADDR AppName, \
    WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, \
    CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, NULL, NULL, \
    hInst, NULL
MOV  hwnd, EAX              ;保存窗口句柄
INVOKE ShowWindow, hwnd, SW_SHOWNORMAL ;显示窗口
INVOKE UpdateWindow, hwnd   ;刷新窗口
;-----
;  进入消息循环
;-----
    . WHILE TRUE
        INVOKE GetMessage, ADDR msg, NULL, 0, 0
        . BREAK . IF (! EAX)
        INVOKE TranslateMessage, ADDR msg
        INVOKE DispatchMessage, ADDR msg
    . ENDW
MOV  EAX, msg.wParam
RET
WinMain ENDP
;-----
;  处理消息的窗口过程
;-----
WndProc PROC hWnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM
    . IF uMsg == WM_LBUTTONDOWN           ;对鼠标左键消息进行处理
        INVOKE MessageBox, NULL, OFFSET szText, OFFSET szCaption, MB_OK
    . ELSEIF uMsg == WM_DESTROY           ;如果用户关闭窗口,则进行退出处理
        INVOKE PostQuitMessage, NULL     ;发出退出程序的消息
    . ELSE

```

```

        INVOKE DefWindowProc,hWnd,uMsg,wParam,lParam;对未处理的消息进行默认处理
        RET
    .ENDIF
    XOR     EAX,EAX                ;正常结束时返回代码为0
    RET
WndProc ENDP
END START

```



图 15-4 程序的执行结果

【程序分析】

(1) 获得应用程序的句柄

程序的第一条语句是调用 `GetModuleHandle` 获得应用程序的句柄。

```

INVOKE GetModuleHandle, NULL
MOV hInstance,EAX

```

Win32 下的应用程序句柄实际上是应用程序在内存中的线性地址。它是唯一的,可以把句柄看成是该应用程序的 ID 号,调用多个 API 函数是把它作为参数来进行传递,所以在一开始便取得句柄并保存在 `hInstance` 全局变量中。

(2) 获得命令行参数

```

INVOKE GetCommandLine
MOV CommandLine,EAX

```

`GetCommandLine` 是一个 API 函数,返回值就是该应用程序的命令行参数。将获得的命令行也保存在一个全局变量 `CommandLine` 中,如果在应用程序中不需要处理命令行参数,则这一步可以省略。

(3) 注册窗口类

如果是用户自定义的窗口,必须通过窗口注册类函数 `RegisterClassEx` 建立自己定义的窗口类。一个窗口类定义了窗口的很多属性。这些属性定义在一个 `WNDCLASSEX` 结构中。`WNDCLASSEX` 的结构定义为:

```

WNDCLASSEX  STRUCT DWORD
cbSize          DWORD    ?          ;WNDCLASSEX 的字节数

```

style	DWORD ?	;从这个窗口类派生的窗口具有的风格
lpfnWndProc	DWORD ?	;窗口过程的地址
cbClsExtra	DWORD ?	;指定紧跟在窗口类结构后的附加字节数
cbWndExtra	DWORD ?	;指定紧跟在窗口事例后的附加字节数
hInstance	DWORD ?	;所属应用程序句柄
hIcon	DWORD ?	;图标句柄
hCursor	DWORD ?	;光标的句柄
hbrBackground	DWORD ?	;背景画刷的句柄
lpszMenuName	DWORD ?	;窗口菜单名字字符串指针
lpszClassName	DWORD ?	;窗口类名称字符串的地址
hIconSm	DWORD ?	;和窗口类关联的小图标句柄
WNDCLASSEX ENDS		

程序在 WNDCLASSEX 结构中填入相应的参数,最重要是将自己定义的窗口过程地址填入 lpfnWndProc 字段;再把结构的地址当参数传递给 RegisterClassEx。

(4) 创建窗口

注册窗口类后,将调用 CreateWindowEx 来创建实际的窗口。该函数的各参数定义如下:

dwExStyle: 附加的窗口风格。通常在 Style 中指定一般的窗口风格,但是一些特殊的窗口风格,如果是顶层窗口则必须在此参数中指定。通常将此参数设为 NULL。

lpClassName: 窗口类名称字符串的地址。

lpWindowName: 窗口名称的地址。该名称会显示在标题条上。

dwStyle: 窗口的风格。在此可以指定窗口的外观。最为普遍的窗口类风格是 WS_OVERLAPPEDWINDOW。

X,Y: 指定窗口左上角的以像素为单位的屏幕坐标位置。默认地可指定为 CW_USEDEFAULT,这样 Windows 会自动为窗口指定最合适的位置。

nWidth, nHeight: 以像素为单位的窗口大小。默认地可指定为 CW_USEDEFAULT,这样 Windows 会自动为窗口指定最合适的大小。

hWndParent: 父窗口的句柄。在父窗口销毁时同时把其子窗口也销毁。在例子程序中因为只有一个窗口,故把该参数设为 NULL。

hMenu: WINDOWS 菜单的句柄。如果只用系统菜单则指定该参数为 NULL。

hInstance: 产生该窗口的应用程序的句柄。

lpParam: 指向欲传给窗口的结构体数据类型参数的指针。一般情况下,该值总为零。

调用 CreateWindowEx 成功后,窗口句柄在 eax 中。必须保存该值以备后用。由于刚刚产生的窗口不会自动显示,所以必须调用 ShowWindow 来显示该窗口。接下来调用 UpdateWindow 来更新客户区。

(5) 消息循环

消息循环代码通常如下:

```
. WHILE TRUE
```

```

    INVOKE GetMessage, ADDR msg, NULL, 0, 0
    .BREAK .IF (! EAX)
    INVOKE TranslateMessage, ADDR msg
    INVOKE DispatchMessage, ADDR msg
.ENDW

```

消息循环不断地调用 GetMessage 从 Windows 中获得消息。GetMessage 传递一个 MSG 结构体给 Windows, 然后 Windows 在该函数中填充有关的消息, 一直到 Windows 找到并填充好消息后 GetMessage 才会返回。在这段时间内系统控制权可能会转移给其他的应用程序。这样就构成了 Win32 下的多任务结构。如果 GetMessage 接收到 WM_QUIT 消息后就会返回 FALSE, 使循环结束并退出应用程序。TranslateMessage 只对键盘消息进行预处理, 从键盘消息中接受按键扫描码并转换成 ASCII 码, 然后在消息队列中插入 WM_CHAR 消息; 其他消息不作任何预处理。最后 DispatchMessage 会把消息发送给负责该窗口过程的函数。当窗口过程返回后, DispatchMessage 函数才返回到消息循环, 然后开始新一轮的消息循环。

在 Windows 下, 消息数据结构定义如下:

MSG STRUCT

hWnd	DWORD ?	;接收该消息的窗口的窗口句柄
message	DWORD ?	;消息的 ID 号, 一般以 WM_ 开头
wParam	DWORD ?	;消息所附带的第一个参数
lParam	DWORD ?	;消息所附带的第二个参数
time	DWORD ?	;消息存入消息队列的时间
pt	POINT <>	;存放消息产生时鼠标的位置(x,y)

MSG ENDS

(6) 窗口过程

窗口过程一般是根据消息标识组成的多路分支程序, 形式如下:

```

WndProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM
    .IF uMsg==WM_X
        处理消息 WM_X
    .ELSEIF uMsg==WM_Y
        处理消息 WM_Y
        :
        :
    .ELSEIF uMsg==WM_DESTROY
        INVOKE PostQuitMessage, NULL
    .else
        INVOKE DefWindowProc, hWnd, uMsg, wParam, lParam
        RET
    .ENDIF
    XOR    EAX, EAX
    RET
WndProc ENDP

```

窗口过程函数的名称可以自己定义,但必须和窗口注册函数里的 `lpfnWndProc` 保持一致。函数第一个参数 `hWnd` 是接收消息的窗口的句柄。`uMsg` 是接收的消息标识,这里的 `uMsg` 不是一个 `MSG` 结构,只是一个 `DWORD` 类型数。`wParam` 和 `lParam` 只是附加参数,以方便传递更多的和该消息有关的数据。

窗口过程函数只对感兴趣的消息加以处理,在例子程序中,对按鼠标左键消息 `WM_LBUTTONDOWN` 进行处理。处理完后,给 `EAX` 寄存器中传递 0,否则必须调用 `DefWindowProc`,把该窗口过程接收到的参数传递给默认的窗口处理函数。

所有消息中必须处理的是 `WM_DESTROY`,当应用程序结束时 Windows 把这个消息传递进来,这仅是通知应用程序窗口已销毁,程序必须自己返回 Windows。在此消息中可以做一些清理工作,处理完清理工作后,必须调用 `PostQuitMessage`,该函数会把 `WM_QUIT` 消息传回应用程序,而该消息会使得 `GetMessage` 返回,并在 `EAX` 寄存器中放入 0,然后会结束消息循环并退回 Windows。

习 题

1. 简述 Win32 汇编语言源程序的结构和特点。
2. 编写 Windows 应用程序时,应该使用何种函数调用方式,为什么要使用这种调用方式?
3. 一个 Windows 程序可以含有几种类型的数据段,这些段的特点是什么?
4. 创建一个名为 `MYSTRUCT` 的数据结构类型,该类型含有二个字段,其中第一个字段 `dwMem` 是 `DWORD` 类型;第二个字段 `bMem` 是含有 5 元素的 `byte` 数组类型,并且它们的初始值都是 `11H`。
5. 用 Win32 汇编中的高级语法编程计算 `sum` 的值, $sum = 1 + 2 + 3 + \dots + 100$ 。
6. 用 Win32 汇编中的高级语法编程实现:假设内存中从 `BUF` 单元开始有若干单字节无符号数,要求把它们按其数值大小,从小到大重新排列。
7. 简述 Win32 汇编环境下的窗口程序的运行过程。
8. 请在例 15.2.1 中调试,程序如果没有调用 `ExitProcess`,将会发生什么样的错误?
9. 编写并运行一个 Win32 汇编语言程序,要求把一个指定的十六进制数转换成对应的字符串,并显示在屏幕上。
10. 编写并运行一个 Win32 汇编语言程序,程序的功能是弹出一个显示“I am a student!”字符串的消息框。

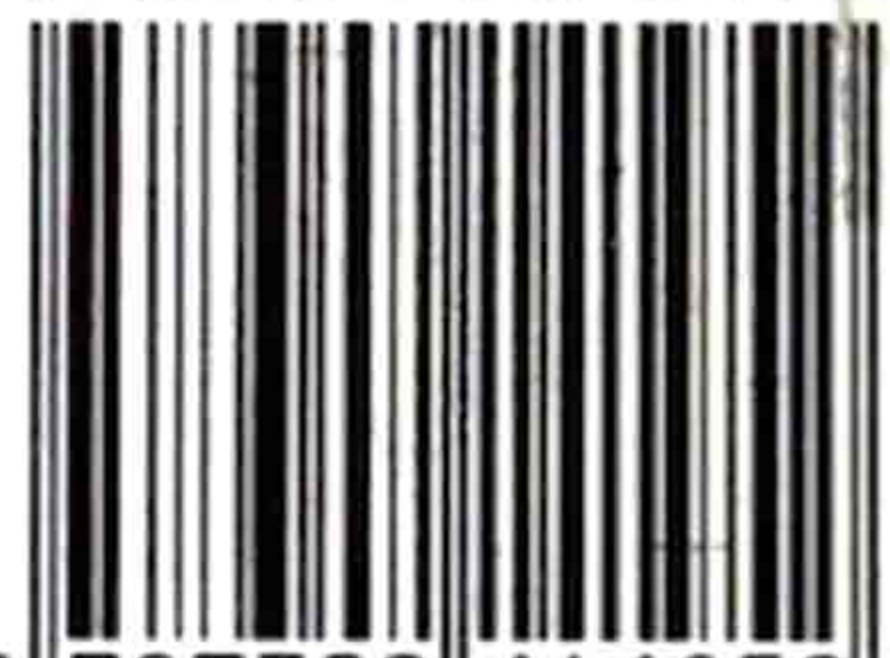
参 考 文 献

1. 杨季文等编著. 80x86 汇编语言程序设计. 北京: 清华大学出版社, 1998
2. 罗云彬编著. Windows 环境下 32 位汇编语言程序设计. 北京: 电子工业出版社, 2002
3. Randall Hyde. Art of Assemble. No Strach, 陈曙晖译. 北京: 清华大学出版社, 2005
4. Iczeliond 的汇编教程, <http://win32assembly.online.fr/>
5. 李彦昌. 80x86 保护模式系列教程, <http://ts.hubce.edu.cn/>
6. Microsoft 的 MSDN 帮助文档, <http://msdn.microsoft.com/>
7. 严义等编著. Win32 汇编语言程序设计教程. 北京: 机械工业出版社, 2004
8. 吴中平编著. Windows 汇编语言程序设计. 北京: 清华大学出版社, 2004
9. 罗省贤等编著. 汇编语言程序设计教程. 北京: 电子工业出版社, 2004
10. 徐建民等编著. 汇编语言程序设计. 北京: 电子工业出版社, 2005
11. 谭毓安等编著. Windows 汇编语言程序设计教程. 北京: 电子工业出版社, 2005
12. 陈建铎等编著. 32 位微型计算机原理与接口技术. 北京: 高等教育出版社, 1998
13. 史新福等编著. 32 位微型计算机原理、接口技术及其应用. 西安: 西北工业大学出版社
14. 宋焕章, 王保恒, 张春元编著. 计算机原理与设计(下册)——存储与外设. 长沙: 国防科技大学出版社, 1999
15. 蒋本珊编著. 电子计算机组成原理. 北京: 北京理工大学出版社, 1999
16. 沈美明, 温冬婵编著. IBM-PC 汇编语言程序设计. 北京: 清华大学出版社, 2000
17. 戴梅萼等编著. 微型计算机技术及应用: 从 16 位到 32 位. 北京: 清华大学出版社, 1996
18. 周明德编著. 微型计算机系统原理及应用. 北京: 清华大学出版社, 2000
19. Intel. Microprocessors Vol. I ~ II. 1993
20. Walter A. Triebel. 80x86/Pentium 处理器硬件、软件及接口技术教程. 王克义等译. 北京: 清华大学出版社, 1998
21. Barry B. Brey. Intel 系列微处理器结构、编程和接口技术大全——80x86、Pentium 和 Pentium Pro. 陈谊等译. 北京: 机械工业出版社, 1998
22. Steven Armbrust Ted Forgeron. DOS/BIOS 使用详解. 舒志勇, 刘东源译. 北京: 电子工业出版社, 1991
23. 范修维. 软盘加密与解密新技术. 北京: 清华大学出版社, 1994
24. 张裁鸿编著. PC 系列机系统开发与应用. 北京: 国防工业出版社
25. 孙德文. 微型计算机技术(修订版). 北京: 高等教育出版社, 2005
26. 戴梅萼, 史嘉权. 微型计算机技术及应用(第 3 版). 北京: 清华大学出版社, 2003
27. 潘新民. 微型计算机硬件技术教程——原理、汇编、接口及体系结构. 北京: 机械工业出版社, 2004

本书特色

- 本书以32位微处理器为背景，讲述微型计算机原理、汇编语言程序设计和接口技术。
- 本书还讲述了保护模式的工作机制，以及Win32汇编语言程序设计的基本方法。
- 本书可作为高等院校计算机专业及电类相关专业本科生“微机原理及应用”、“汇编语言程序设计”、“微机原理与接口技术”等课程的教材和参考书。
- 本书比较详细地讲述了中断系统和串行通信，对于从事微机系统应用与开发的工程技术人员有一定的参考价值。

ISBN 978-7-302-14195-2



9 787302 141952 >

定价：38.00元